NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A COMPARISON OF
SOME OF THE MOST CURRENT METHODS OF
IMAGE COMPRESSION

by

Robert T. Kay

June, 1993

Thesis Advisor: Ron J. Pieper

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1a Report Security Classification: Unclassified		1b Restrictive Markings	
2a Security Classification Authority		3 Distribution/Availability of Report	
2b Declassification/Downgrading Schedule		Approved for public release; distribution is unlimited.	
4 Performing Organization Report Number(s)		5 Monitoring Organization Report Number(s)	
6a Name of Performing Organization Naval Postgraduate School	6b Office Symbol *(if applicable)* EC	7a Name of Monitoring Organization Naval Postgraduate School	
6c Address *(city, state, and ZIP code)* Monterey, CA 93943-5000		7b Address *(city, state, and ZIP code)* Monterey, CA 93943-5000	
8a Name of Funding/Sponsoring Organization	6b Office Symbol *(if applicable)*	9 Procurement Instrument Identification Number	
Address *(city, state, and ZIP code)*		10 Source of Funding Numbers	
		Program Element No	Project No / Task No / Work Unit Accession No

11 Title *(include security classification)* A COMPARISON OF SOME OF THE MOST CURRENT METHODS OF IMAGE COMPRESSION

12 Personal Author(s) Kay, Robert T.

13a Type of Report	13b Time Covered From To	14 Date of Report *(year, month, day)*	15 Page Count
Master's Thesis		June 1993	85

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms *(continue on reverse if necessary and identify by block number)*
Field	Group	Subgroup	Fractal Image Compression, JPEG Image Compression

19 Abstract *(continue on reverse if necessary and identify by block number)*
In this report, commonly used lossless and lossy image compression algorithms are heuristically presented and then compared in terms of performance. The lossy algorithms, JPEG (Joint Photographic Experts Group) and Fractal compression, are compared in terms of their respective sensitivities between compression ratio and image fidelity. Compression algorithms based on the lossless models of Huffman, Adaptive Huffman, and Arithmetic coding are compared in terms of compression ratio and compression/decompression time requirements. High fidelity image reconstructions of JPEG and Fractal compressions are also included in the comparison. Results, for the images tested, indicate that if imperceptible losses in fidelity can be tolerated, then among the current versions of the algorithms tested, the JPEG results in higher compression with less process time.

20 Distribution/Availability of Abstract	21 Abstract Security Classification	
__ unclassified/unlimited __ same as report __ DTIC users	Unclassified	
22a Name of Responsible Individual Pieper, R.J.	22b Telephone *(include Area Code)* (408) 656-2101	22c Office Symbol EC/Pr

DD FORM 1473,84 MAR 83 APR edition may be used until exhausted security classification of this page
All other editions are obsolete Unclassified

Approved for public release; distribution is unlimited.

A Comparison of
Some of the Most Current Methods of
Image Compression

by

Robert T. Kay
Lieutenant, United States Navy
B.S., United States Naval Academy, 1987

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1993

Author: Robert T. Kay
Robert T. Kay

Approved by: Ron J. Pieper
Ron J. Pieper, Thesis Advisor

Ismor Fischer
Ismor Fischer, Second Reader

M. A. Morgan
Michael A. Morgan, Chairman
Department of Electrical & Computer Engineering

ABSTRACT

In this report, commonly used lossless and lossy image compression algorithms are heuristically presented and then compared in terms of performance. The lossy algorithms, JPEG (Joint Photographic Experts Group) and Fractal compression, are compared in terms of their respective sensitivities between compression ratio and image fidelity. Compression algorithms based on the lossless models of Huffman, Adaptive Huffman, and Arithmetic coding are compared in terms of compression ratio and compression/decompression time requirements. High fidelity image reconstructions of JPEG and Fractal compressions are also included in the comparison. Results, for the images tested, indicate that if imperceptible losses in fidelity can be tolerated, then among the current versions of the algorithms tested, the JPEG results in higher compression with less process time.

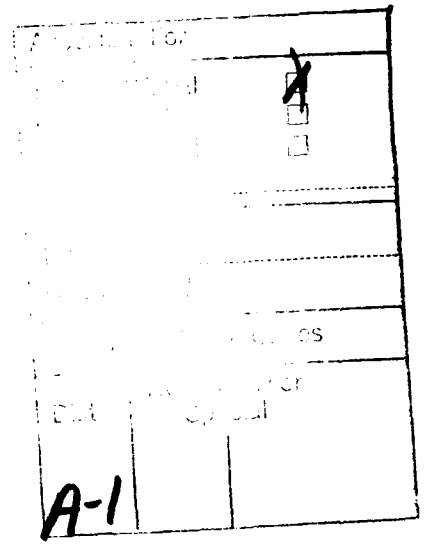

TABLE OF CONTENTS

I. INTRODUCTION

A. REVIEW OF LITERATURE

The explosive proliferation of information in recent years has created a significant demand for the efficient storage, access, and transmission of this data. This is especially true for digital images, as an extremely large number of bits is required in order to represent even a modestly sized single image with acceptable quality and resolution. Image compression, which has come into existence only within the last ten years, is the area of image processing that deals with this problem (Jain, 1981), (Nelson, 1992), (Rabbani, 1991). Its goal is to reduce the number of bits used to store or transmit the image, yet retain an acceptable quality for the end user.

A variety of image compression techniques have been developed over the years. They have been based on lossless and lossy properties. The methods making use of lossless properties generate an exact duplicate of the original image upon decompression. Lossy methods, on the other hand, relinquish some accuracy in exchange for increased compression. The efficiency of each of these compression algorithms is measured by its compressing ability, distortion or fidelity between the original and final decompressed image, and

computational complexity, which has a direct relation to time requirements for implementation (Jain, 1981, p. 349).

Some of the lossless algorithms that have been developed are the Huffman and Adaptive Huffman (Knuth, 1985), (Nelson, 1992), (Rabbani, 1991), Arithmetic (Langdon, 1984), (Nelson, 1992), (Rissanen, 1979), (Witten, 1987), and the Ziv and Lempel (LZ78) models (Jackson, 1993), (Nelson, 1992). With the applications for image processing growing dramatically (for instance in satellite imaging, computer graphics in advertising and entertainment, and model simulation in science and engineering), lossy compression techniques have received the most attention in recent years. One widely accepted standard is the Joint Photographic Experts Group - Discrete Cosine Transform (JPEG-DCT) (Ahmed, 1974), (Ahmed, 1975), (Nelson, 1992), (Wallace, 1992). Additionally, a relatively new method being explored takes advantage of the fractal character for compression of an image. It makes use of iterative techniques to exploit the redundancy in images (Barnsley, 1993), (Fisher, 1992), (Jacobs, 1992), (NOSC TR1315), (NOSC TR1362), (NOSC TR1408).

B. OVERVIEW OF THE THESIS

The current chapter introduces the area of image processing known as image compression. The various methods of image

compression and the basis for determining the efficiency of each are presented.

Chapter II discusses the models of three accepted lossless compression techniques whose efficiencies will be examined. Those models discussed for future comparison are the Huffman, Adaptive Huffman, and Arithmetic algorithms.

Chapter III describes the methods of the two lossy compression techniques that will be analyzed in this research. The lossy routine models presented are the widely utilized JPEG-DCT and the relatively new Fractal-based algorithm.

In Chapter IV, a comparative analysis of the different efficiencies of each of the presented image compression techniques is performed. The advantages and disadvantages of each of the examined methods are discussed.

The general conclusions reached from the comparative analysis of Chapter IV are presented in Chapter V.

Appendix A covers the operational mechanics which were required to gather data for comparison of the different image compression techniques. Topics discussed include image format, conversion between image formats, display of images, and PC versus Sun Workstation operations. Appendix B lists some public domain Fractal Compression Code and Appendix C contains the numerical data gathered during the research.

II. LOSSLESS COMPRESSION TECHNIQUES

A. THE HUFFMAN ALGORITHM

The basic premise of Huffman coding is the creation of variable-length codes for each symbol, with each code being represented by an integral number of bits. Symbols with higher probabilities are assigned shorter bit codes while symbols with lower probabilities are assigned longer bit codes. Once the frequency or probability for every symbol in a source is determined, the Huffman code can be constructed by repeatedly combining the two least probable symbols at each stage until the original source is reduced to only two symbols. These two symbols are respectively assigned the bit values of '0' and '1'. The codes for the previous reduced stage are then determined by appending a '0' or '1' to the right of the code corresponding to the two least probable symbols. The process is repeated until each symbol in the original source is assigned a code, thus obtaining the Huffman code. Table II.1 shows an example source reduction and Table II.2 performs the resulting codeword construction for generating the Huffman code. Looking at Table II.2, it can be seen that the final codes have the unique prefix property, meaning no single code is a prefix for another code. Therefore, they can be unambiguously decoded as they arrive in a continuous

TABLE II.1: EXAMPLE SOURCE REDUCTION PROCESS FOR HUFFMAN CODING.

Original Source				Reduced Source		Reduced Source		Reduced Source
s_i	Count	Prob						
s_1	20	0.40	s_1	0.40	s_1	0.40	s_{2345}	0.60
s_2	10	0.20	s_2	0.20	s_{345}	0.40	s_1	0.40
s_3	10	0.20	s_3	0.20	s_2	0.20		
s_4	6	0.12	s_{45}	0.20				
s_5	4	0.08						

TABLE II.2: EXAMPLE SOURCE CODEWORD CONSTRUCTION PROCESS FOR HUFFMAN CODING.

Original Source			Reduced Source		Reduced Source		Reduced Source
s_i	Codeword						
s_1	1	s_1	1	s_1	1	s_{2345}	0
s_2	000	s_2	01	s_{345}	00	s_1	1
s_3	001	s_3	000	s_2	01		
s_4	010	s_{45}	001				
s_5	011						

stream. Additionally, the symbol with the highest probability, s_1, has been assigned the fewest bits while the symbol with the lowest probability, s_5, has been assigned the greatest bits. It should be noted that symbols of equal weight can be interchanged to make an equally optimal Huffman code.

Each of the above observations contribute to make Huffman coding fairly simple to implement.

One of the limitations of Huffman coding is that since the number of bits for each code must be an integer, the ideal code length for a symbol is met only when its probability is a negative power of two, i.e., 1/2, 1/4, 1/8, etc. This is because the ideal binary code length for a symbol s_i is $l(s_i) = -\log_2(p_i)$, where p_i is the probability of s_i. Therefore, the chance of the Huffman code being set to ideal lengths is not very likely. The example in Tables II.1 and II.2 accomplishes compression by reducing the average symbol length (L_{avg}) from 3.0 to 2.0.

$$L_{avg} = \sum_{i=1}^{n} p_i \, l(s_i) \qquad (II.1)$$

where n is the number of symbols. The original L_{avg} is 3.0 because three binary bits are needed to differentiate between five symbols. Another limitation is that a copy of the probability table must be transmitted with the compressed data since the expansion program would otherwise not be able to decode it correctly. A preset Huffman code could be used to avoid this limitation, but then the model is not very adaptable to changing source statistics. In fact, there is even the possibility of expansion if the preset code is used with changing sources (Rabbani, 1991, pp. 27-28).

B. ADAPTIVE HUFFMAN ALGORITHM

In the Huffman model discussed above, the probability of each of the symbols was determined without any consideration of the symbols that preceded it. This is known as a zero order model. By accounting for predictability, or increasing the model order, one may further reduce the number of bits required for the data. The trade-off though, is that the number of probability tables that must be transmitted with it will also increase. In essence, the savings in image data are negated by the requirement for additional probability tables.

Adaptive Huffman coding allows for the use of higher order modeling without the requirement of the added probability tables. This is accomplished by adjusting the Huffman codes progressively, based only on previous data. Instead of first determining probabilities and then encoding as in the Huffman procedure, the Adaptive model initially assumes all symbol weights are zero and counts the symbol frequencies as it encodes them. After each symbol, the Huffman code is modified to account for the new character. The decoding process similarly learns the symbol frequencies and modifies the Huffman code in the same fashion. Thus, the encoder and decoder remain synchronized because any changes to symbol probabilities in the encoder are also taking place in the decoder. The only requirement is that both sender and receiver know the size of the symbol domain, which is the

number of different symbol possibilities (Knuth, 1985, pp. 163-164).

C. ARITHMETIC ALGORITHM

Even though variations of Huffman coding are currently accepted to be the most efficient fixed-length lossless coding methods, they still have one major disadvantage. This is the requirement that symbol codes be an integral number of bits. As stated earlier, this only occurs for probabilities of 1/2, 1/4, 1/8, etc. If the probability of a symbol is 1/5, the optimum code length would be $-\log_2(0.2) = 2.32$ bits. Huffman code would require two or three bits to encode the symbol, thus preventing maximum compression.

A viable solution to this problem is Arithmetic coding, which is another lossless technique that represents the entire message as a number stream. The idea is to represent the entire symbol domain as the interval of real numbers between zero inclusive and one exclusive ([0, 1)). Each symbol, based on its probability, is assigned a range within the interval. Table II.3 demonstrates a sample interval range assignment.

Before encoding is initiated, the range is [0, 1). As each symbol is processed, the range is narrowed to that interval within the current range which is allocated to the symbol. As successive symbols are processed, the interval becomes smaller and smaller. The higher the probability of a

TABLE II.3: EXAMPLE ARITHMETIC CODING RANGE ASSIGNMENT.

Symbol	Probability	Range
A	0.40	[0.00, 0.40)
B	0.25	[0.40, 0.65)
C	0.15	[0.65, 0.80)
D	0.10	[0.80, 0.90)
E	0.10	[0.90, 1.00)

symbol, the less it will reduce the range and therefore, add fewer bits to the code. Table II.4 shows the process based on the symbol probabilities listed in Table II.3.

TABLE II.4: ARITHMETIC ENCODING PROCESS.

Symbol Number	Symbol	Low Value	High Value
1	B	0.40	0.65
2	A	0.40	0.50
3	D	0.48	0.49
4	A	0.480	0.484
5	C	0.4826	0.4832
6	E	0.48314	0.48320

The decoding process is then fairly straightforward. The first symbol is determined from the subinterval of [0,1) in which the encoded message falls. The next symbol is discovered by subtracting the low value of the first symbol with the encoded value and dividing by the width of the range. The

symbol is then found via the interval in which the new encoded value falls. The decoding algorithm for the message "BADACE" is illustrated in Table II.5.

TABLE II.5: ARITHMETIC DECODING PROCESS.

Encoded Number	Symbol	Low	High	Range
0.48314	B	0.40	0.65	0.25
0.33256	A	0.00	0.40	0.40
0.8314	D	0.80	0.90	0.10
0.314	A	0.00	0.40	0.40
0.785	C	0.65	0.80	0.15
0.90	E	0.90	1.00	0.10

It should be noted that in actual coding, the values of the encoded numbers will be represented in binary. See Langdon (1984, pp. 136-139) for an example utilizing binary values. Decimal values were utilized in the above example to assist the reader in understanding the concept (Nelson, 1992, p. 128). Since the decoder interprets the encoded number 0.0 as a symbol (A in Table II.3) in the domain interval, an end of message symbol known to both the encoder and decoder is required (Witten, 1987, p. 522).

An example by Nelson (1992, p. 133) provides insight into how compression is obtained in Arithmetic coding. Assume a stream 'AAAAAAA' is to be compressed. The probability of A is known to be 0.9 while the end-of-message character has a

probability of 0.1. The ranges [0, 0.9) and [0.9, 1.0) are assigned to the A and end-of-message characters respectively. Table II.6 shows the results.

TABLE II.6: SAMPLE ARITHMETIC ENCODING TO SHOW COMPRESSION.

New Character	Low Value	High Value
A	0.0	0.9
A	0.0	0.81
A	0.0	0.729
A	0.0	0.6561
A	0.0	0.59049
A	0.0	0.531441
A	0.0	0.4782969
END	0.43046721	0.4782969

One dilemma of Arithmetic coding is that most computers cannot process numbers of the length needed to encode an image. This is corrected by using an incremental transition scheme which links the high and low end bits of successive numbers in the symbol stream. Another problem is that of loss of precision between the high and low values as the range gets very small. This can result in the low value being higher than the high value and consequently, causing underflow. This is eliminated by inserting checks in the process and so increased compression is achieved at the expense of increased complexity.

Arithmetic coding has shown the most promise in compression of black and white or two value, one bit per pixel images (Langdon, 1984, pp. 140-142), (Langdon, 1981, pp. 863-866). Huffman coding as performed in Chapter II.A is unable to compress these images due to the integral coding requirement. Lossy techniques have also proven unrealistic because a loss of quality in decompression can result in the opposite color being output, which could drastically degrade the final image due to there being only two colors.

III. LOSSY COMPRESSION TECHNIQUES

A. JPEG-DCT ALGORITHM

With the ongoing advances in digital image technology, it was realized quite some time back that the previously mentioned lossless compression techniques were not going to be satisfactory due to the enormous amounts of data required to display a digital image. For example, a digitized, single image at color television quality requires upwards of one million bytes or 8 million bits of data storage. Therefore, in 1989, the ISO and CCITT (International Standards Organization/Consultive Committee for International Telephone and Telegraph) joined together to form the JPEG committee to set an image compression standard (Wallace, 1992, p. xix). Though the JPEG standard has not yet been officially published, it is near enough to its final stages so that its applications are now being widely used in commercial applications.

The JPEG encoder, its model shown in Figure III.1, achieves compression with the combination of lossy quantization followed by entropy encoding. In the most common form of the JPEG algorithm, the entropy process is carried out by the Huffman or Arithmetic methods (Wallace, 1992, p. xxiii). The quantization step allows the user to sacrifice quality in order to achieve greater compression. For decompression, the

JPEG decoder performs the same steps in reverse order, with one slight alteration - an IDCT (Inverse Discrete Cosine Transform) replaces the DCT process.

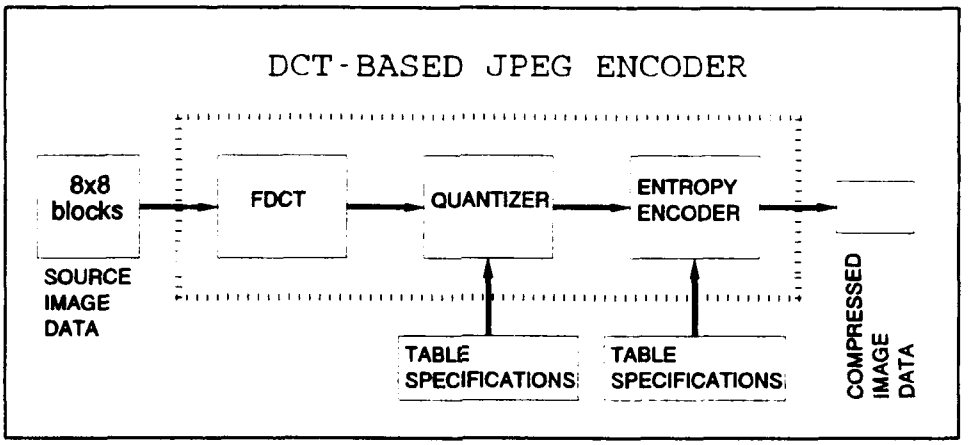

Figure III.1: DCT-Based Encoder Processing Steps (Wallace, 1992, p. xxi).

The first of the four main steps for compression is the partitioning of the input data into groups of 8x8 pixels in preparation for the DCT (Discrete Cosine Transform), which is performed in the second step. The reason for the 8x8 grouping is that a DCT performed over the entire image would require an inordinate number of computations, as can be seen by the following equations:

$$DCT(i,j)=\frac{1}{\sqrt{2N}}C(i)C(j)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}pixel(x,y)\cos[\frac{(2x+1)i\pi}{2N}]\cos[\frac{(2y+1)j\pi}{2N}] \qquad (III.1)$$

$$where:\ C(i),\ C(j) = \frac{1}{\sqrt{2}}\ for\ i,j=0,\ else\ C(i),\ C(j) = 1.$$

In the case of the JPEG, N=8. In the decoding portion, the IDCT is defined as follows:

$$IDCT(x,y) = \frac{1}{\sqrt{2N}} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} C(i)C(j)DCT(i,j)\cos[\frac{(2x+1)i\pi}{2N}]\cos[\frac{(2y+1)j\pi}{2N}] \quad (III.2)$$

The justification for choosing 8x8 sized blocks vice 16x16 or any other size is that research has shown there is very little predictability between pixels spaced more than eight positions away (Nelson, 1992, P. 360).

There are two reasons for the general use of the DCT/IDCT versus the DFT/IDFT (Discrete Fourier Transform/Inverse Discrete Fourier Transform). See Gonzalez (1987, pp. 65-69) for a definition and explanation of the DFT/IDFT. The primary reason is that during reconstruction of the individual blocks, pixel disparities on opposite sides of the boundaries cause the DFT to leave edge artifacts at the boundaries. In an image divided into numerous 8x8 blocks, these boundary discontinuities would be highly visible and thus, unacceptable (Rabbani, 1991, pp. 109-110). As it turns out, it can be shown that an N-point DCT can be represented as the real part of the 2N-point DFT of a data sequence or pixel set whose values at N+1, N+2, ..., 2N are equal to zero (Bracewell,

1986, pp. 17-18). This is equivalent to the 2N-point DFT of a sequence in which the pixel values from points [0, N-1] are reflected about a vertical axis placed at N and repeated to form an even periodic data set. Due to the smoothness of the data set at the boundary, upon reconstruction of the image there will not be pixel value disparities or edge artifacts at the exterior points. For the second argument, there is little sense in taking the DFT and then discarding the imaginary values while retaining the real values, when the DCT can perform the same function in one step with half the computations. Despite this solution, some residual edge artifacts from the DCT are still known to be generated after quantization (Rollins, 1992, pp. 191-199).

Upon completing the DCT transformation, the data must then be quantized. Quantization is vital to obtain compression because the DCT is a lossless transformation that does not actually compress the image data. Instead, it concentrates the majority of the information into a few coefficients in the upper left-hand corner of the data block, the importance of which will be explained later.

Quantization achieves the majority of the compression by modifying the DCT transformed coefficients into values requiring fewer bits to represent. It is accomplished by dividing each DCT coefficient by the corresponding quantizing value and rounding to the nearest integer.

$$QuantizedValue = INTEGER\{\frac{DCT(i,j)}{Q(i,j)}\} \qquad (III.3)$$

Like the data processed by the DCT, the quantization table is an 8x8 block, but it is specified by the user, who makes a choice based on the desired final image quality. It is effective because it forces many of the DCT coefficients to truncate to zero. These zero coefficients are not overly important to the reconstructed image quality because after the DCT transformation, as stated earlier, most of the useful information is concentrated in the upper left-hand corner (0,0 position) of the DCT block. This coefficient is an average value of the overall magnitude of the input data and is called the DC coefficient.

Prior to the final step, the quantized values are arranged in a zig-zag sequence (see Figure III.2) to organize the data so that the zero values are placed in a more efficient

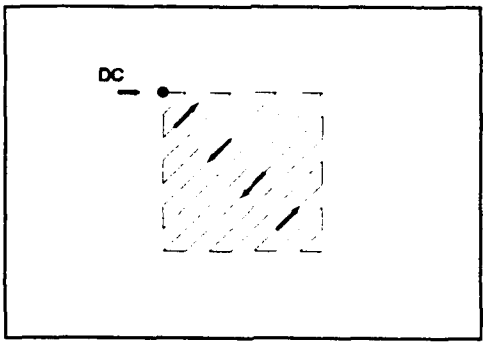

Figure III.2: Zig-zag sequencing which is performed after quantization.

consecutive ordering. With the zero values arranged in this continuous fashion, it is possible to achieve further compression by using a lossless technique such as Huffman or Arithmetic coding (Wallace, 1992, p. xxiii). In Nelson's (1992) simplified version of the JPEG algorithm, the scheme utilizes runlength encoding in place of the more standard choices mentioned above.

Some steps of the JPEG compression technique will now be demonstrated in some examples taken from Nelson (1992, pp. 365-368). In each case, the image is 8x8 pixels and 256 different grey-scale colors. Figure III.3 shows a sample data-bit representation of a non-quantized test image before and after processing by the DCT algorithm.

The values of the DCT output warrants some discussion. "Because of the DCT's application importance and its relationship to the DFT, many different algorithms by which the DCT and IDCT may be approximately computed have been devised." (Wallace, 1992, p. xxi). Small variations in implementation or precision may cause different output for the same input. Additionally, there are varying methods to input and store the output data. As seen in Figure III.3, DCT output values can be negative, and in the case of the DC coefficient, greatly increased in magnitude. Nelson (1992, pp. 364-365) deals with this obstacle in the simplest fashion by allowing 11 bits per value. He assumes they may vary from -1,024 to 1,023. More

Input Pixel Values:

140	144	147	140	140	155	179	175
144	152	140	147	140	148	167	179
152	155	136	167	163	162	152	172
168	145	156	160	152	155	136	160
162	148	156	148	140	136	147	162
147	167	140	155	155	140	136	162
136	156	123	167	162	144	140	147
148	155	136	155	152	147	147	136

Output Pixel Values:

186	-18	15	-9	23	-9	-14	19
21	-34	26	-9	-11	11	14	7
-10	-24	-2	6	-18	3	-20	-1
-8	-5	14	-15	-8	-3	-3	8
-3	10	8	1	-11	18	18	15
4	-2	-18	8	8	-4	1	-7
9	1	-3	4	-1	-7	-1	-2
0	-8	-2	2	1	4	-6	0

Figure III.3: Sample image data before and after processing by the DCT.

memory is required than the original eight-bit values, but the quantization and entropy compression steps easily offset this temporary increase. Rabbini (1991, pp. 114-115) uses only eight bits, but stores the data based on a range and its difference from previous values. Wallace (1992, p. xx) shifts the input to the DCT from $[0, (2^p-1)]$ to $[-(2^{p-1}), (2^{p-1}-1)]$, with p=8 in this instance. Each method has its advantages and disadvantages based on the complexity of implementation and storage requirements - variables every user must consider.

As mentioned before, quantization is a user-selected variable. In the next example, the quantization table chosen can be seen in Figure III.4. The effects of this quantization block on a sample DCT transformed image can be seen in Figure III.5. By then reordering into the zig-zag sequence previously shown in Figure III.2, further compression of the data can be accomplished by processing it through a lossless encoder.

3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

Figure III.4: Selected Quantization Table (Nelson, 1992, p. 367)

B. FRACTAL COMPRESSION

1. Conceptual Background

Another lossy method of compression is based on the "self-similarities" or "fractals" inherently present in an image. When magnified, a small portion of an image may closely resemble a larger portion of the same image. Benoit Mandelbrot, considered to be the father of fractal theory, demonstrated that random, computer generated fractals could produce

DCT Before Quantization:

92	3	-9	-7	3	-1	0	2
-39	-58	12	17	-2	2	4	2
-84	62	1	-18	3	4	-5	5
-52	-36	-10	14	-10	4	-2	0
-86	-40	49	-7	17	-6	-2	5
-62	65	-12	-2	3	-8	-2	0
-17	14	-36	17	-11	3	[illegible]	1
-54	32	-9	-9	22	0	1	3

DCT After Quantization:

90	0	-7	0	0	0	0	0
-35	-56	9	11	0	0	0	0
-84	54	0	-13	0	0	0	0
-45	-33	0	0	0	0	0	0
-77	-39	45	0	0	0	0	0
-52	60	0	0	0	0	0	0
-15	0	-19	0	0	0	0	0
-51	19	0	0	0	0	0	0

Figure III.5: A sample DCT transformed image before and after quantization (Nelson, 1992, p. 368)

realistic representations of clouds, coastlines, trees, etc.

> Because computer-generated fractal images have similar patterns on many different scales, relatively little code is all that is usually needed to create them. Once written to produce the detail on one scale, much the same software can be reused in a loop to repeat the image on successively larger (or smaller) scales. Thus a remarkably intricate image blossoms from a small, simple piece of software (Zorpette, 1988, p. 29).

In 1988 Barnsley proposed that if high quality images could be created from only a few initial parts, then it should be possible to reverse the process in order to gain comparable compression. With his proposal though, he realized that applying the reverse technique to real images, as opposed to

generating selectively redundant pictures, was going to be a substantially more complex procedure. He has since shown that recursive iteration $x_{n+1}=W(x_n)$ of an initial image x_0 under a collection W of carefully chosen affine transformations[1] converges toward a desired image p, referred to as an "attractor". The technique is known as an Iterated Function System (IFS). Essentially, Barnsley applied the Contraction Mapping Theorem to images for the purpose of data compression. In general the theorem states if a mapping, W, is in fact "contracting", then iteration of any data set x_0 under W will converge to a unique point p that remains fixed, i.e. $W(p)=p$ (Barnsley, 1993, pp. 70-73). The information represented by this attractor can be encoded in the coefficients of the associated affine transformations. Figure III.6 demonstrates this principle for two different images. Images III.6(a) and III.6(b) are reduced by a half and reproduced three times. By the third iteration of this transformation, both images show definite convergence to the attracting "fixed point" image in Figure III.6(c).

2. An Algorithm for Fractal Compression/Decompression

The description to follow is based on fundamental principles for Fractal compression (Barnsley, 1993), (Fisher,

[1]The general affine transformation is a linear transformation followed by a translation. For the purpose of Fractal compression the coefficients of the linear transformation are constant (Barnsley, 1993, p. 52).

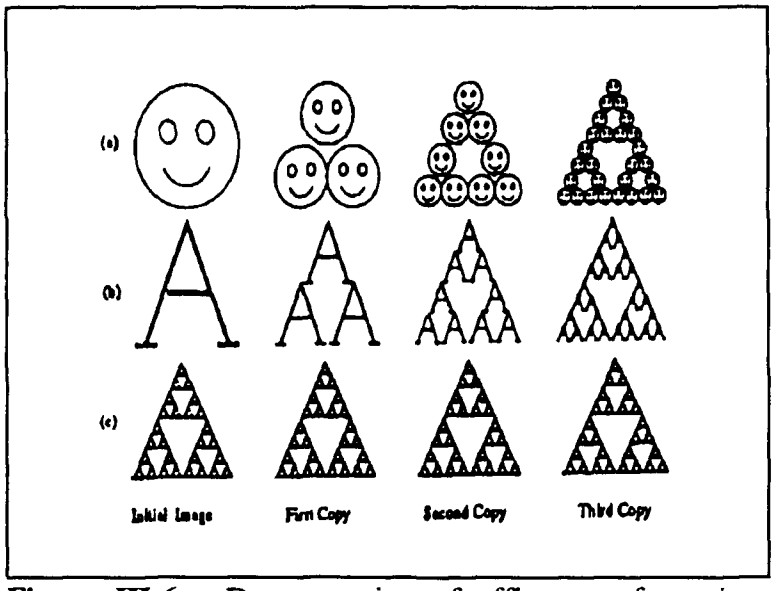

Figure III.6: Demonstration of affine transformations causing an image to converge to an attractor (Fisher, 1992, p. 2).

1992). A public domain program written in C, which is built on these principles (Young, 1992), can be found in Appendix B.

Consider an image of 256x256 pixels with 256 possible grey-scale levels. The image is first divided up into non-overlapping blocks of 8x8 pixels called ranges. There are a total of 1,024 of these ranges. The image is also divided into individual domains, which comprise all possible 16x16 pixel blocks in the image. This comes to 241 times 241, or 58,081 total domains. For each range, the domains are searched for a match that bears a likeness to the portion of the image above that range. During this search, the domain

square is be flipped and rotated to find the best match. Since it is the matching of two squares, there are 8 possibilities for each domain: rotations of 0, 90, 270, and 360 degrees and flipping about the vertical, horizontal, and both diagonal axes. To account for the area of the domain being four times that of the range, a subsample or average of each 2x2 pixel area is taken for comparison purposes. The point of this matching is to find the "best" affine transformation for each domain:

$$w\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & s \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} e \\ f \\ o \end{bmatrix} \qquad (III.4)$$

where s controls the contrast, o controls brightness, and the remaining variables a, b, c, d, e and f determine how the partitioned domain is mapped to the range. The least-mean-squares value is used to determine if a mapping is acceptable.

$$R=\sum_{i=1}^{n}(s\cdot a_i + o\cdot b_i)^2 \qquad (III.5)$$

But first, the range and domain values for the possible mapping are used to calculate the contrast and brightness variables.

$$s = \frac{[n^2(\sum_{i=1}^{n} a_i b_i) - (\sum_{i=1}^{n} a_i)(\sum_{i=1}^{n} b_i)]}{[n^2 \sum_{i=1}^{n} a_i^2 - (\sum_{i=1}^{n} a_i)^2]} \qquad (III.6)$$

$$o = \frac{[\sum_{i=1}^{n} b_i - s \sum_{i=1}^{n} a_i]}{n^2} \qquad (III.7)$$

where n is the number of pixels in the domain or range (64 in this case), and the a_i's and b_i's are respectively the domain and range pixel values. By substituting the results from formulas (III.6) and (III.7) into formula (III.5), a least-mean-squares value will be available for comparison. If this value is less than the initial user input, then the mapping is acceptable. If it does not meet the specification, the search continues on to the next domain transformation, or in the case that all eight, potential transformations have been attempted, the next domain. If every domain has been searched without meeting the least-mean-squares specification, the best possible mapping of all domain transformations is selected.

After a mapping transformation is obtained for all 1,024 ranges, the information can be encoded. The encoding requires 16 bits for the position of the 16x16 pixel domain, seven for the brightness, five for the contrast, and three for the flip/rotation needed to map the domain to a range. The

bit requirements for the contrast and brightness were based on the desired accuracy (NOSC TR1408, p. 13). With these bit requirements , the original image of 65,536 bytes (256x256x1) can be compressed to 3,968 bytes (31/8x1024), which is a compression ratio of 16.52:1. Figure III.7 is a flowchart of the Fractal compression procedure.

Decoding is implemented with the following formula:

$$p \approx p' = W(p') \approx W(p) = w_1(p) \bigcup w_2(p) \bigcup \ldots w_N(p) \qquad (III.8)$$

where p is the original image, p' is the decompressed or transformed image, and W is the combination of individual affine transformations w_i (see formula III.4) which converges to the fixed point image p'. In essence, arbitrary values are input to the domain and iteratively processed through the transform equation. As the process is repeated, the arbitrary domain values progress towards their respective fixed points, which together form the approximation p' of the original image p.

The example provided is a rather simple analysis of the fractal compression algorithm. There are a number of techniques, some implemented and some proposed, that can greatly increase speed and especially, compression (Fisher, 1992, pp. 16-20), (Barnsley, 1993, pp. 119-171). They include varying the size and shape of the range and also classifying

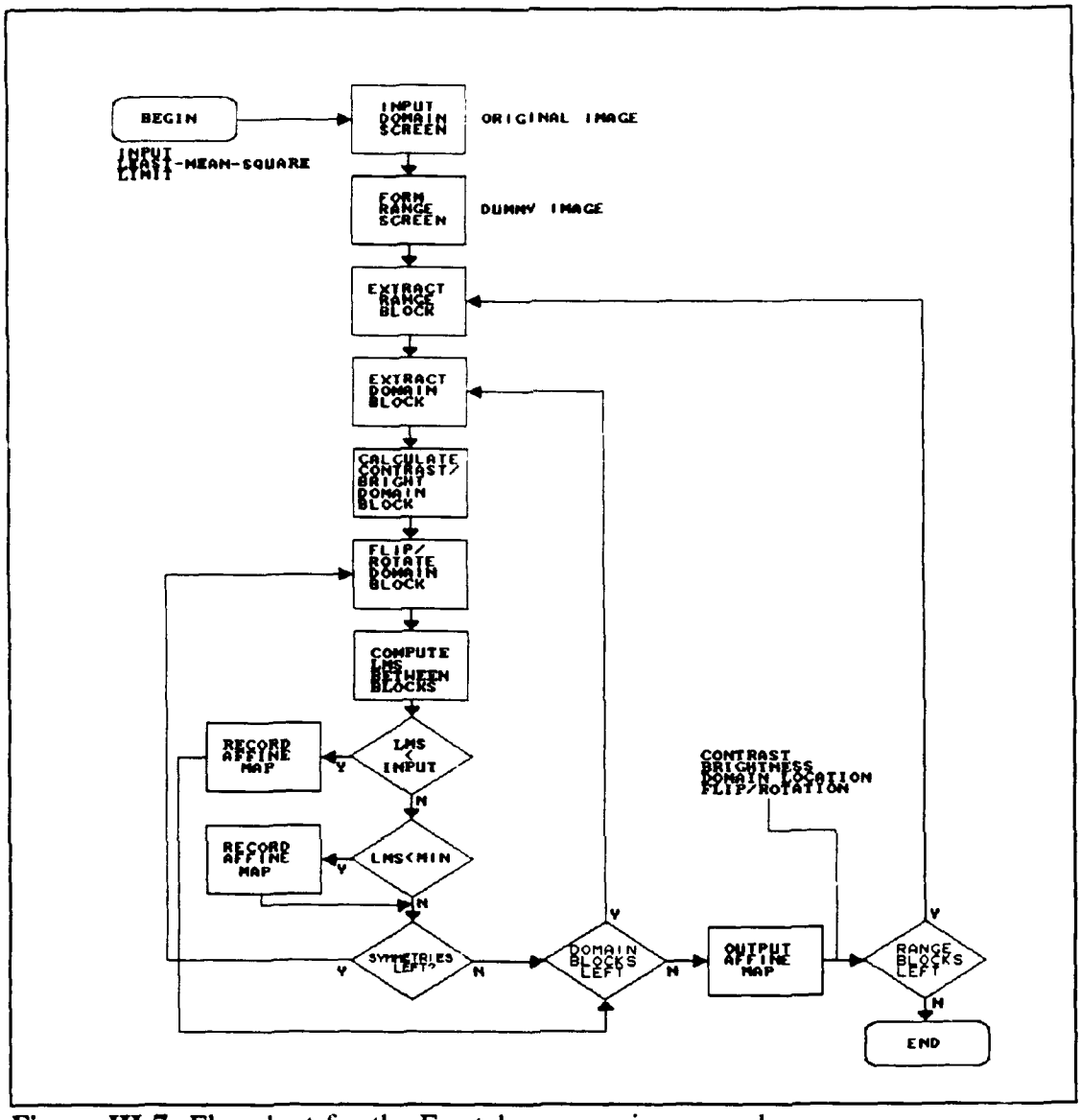

Figure III.7: Flowchart for the Fractal compression procedure.

the ranges and domains based on possible similarities. In this way, the program does not check domains it knows will not converge towards a specified range.

The trade-off for speed and compression depends on user requirements. By increasing or making the least-mean-

square specification less restrictive, speed and compression can be increased, but the quality of the decompressed image is reduced. Since arbitrary data is used for the initial decompressed image, the final decompressed image is only as good as the affine transformation chosen for recursive iteration. The affine transformation, in turn, is then only as good as the least-mean-square restriction selected by the user.

IV. COMPARISON OF METHODS

A. OVERVIEW

In terms of performance, the lossless and lossy compression techniques discussed in Chapters II and III are analyzed for compression ratio, fidelity, and compression/decompression time requirements. The compression ratio is the original image memory size divided by the compressed data memory size. The fidelity is a measure of the quality between the original image and the reconstructed image. For the purposes of this research, the root-mean-square error (rms) is used to evaluate the error between the two images (Gonzalez, 1987, pp. 256-257).

$$e_{rms} = \frac{1}{N}\left[\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} |g(x,y) - f(x,y)|^2\right]^{1/2} \qquad (IV.1)$$

For NxN pixel images, f(x,y) consists of the individual pixel values for the original image and g(x,y) consists of the individual pixel values for the reconstructed image.

With the exception of Fractal compression/decompression, the programs utilized in the comparison are taken from Nelson's (1992) text. The Fractal compression/decompression program, which is not in public domain, is provided for the study and is in executable form only (Netrologic, Inc., 1993).

The basic principles for these Fractal programs are outlined in Fisher's notes (SIGRAPH, 1992). The public domain compression/decompression C language programs in Appendix B (Young, 1992) are built upon the same principles. It is noted that the public domain version in Appendix B, which is not used in the comparison, runs significantly slower.

B. PRESENTATION OF DATA

Three different images were selected to be processed by each of the compression algorithms. These images can be seen in Figure IV.1. Each image is 256x256 pixels with 256 possible shades of gray. The format used for input into the compression programs is raw pixel grey map. In other words, eight bits are needed per pixel so as to distinguish between the possible grey-scales, which are represented in memory by a symbol from the 256 ASCII character set. The data is read proceeding from left to right and top to bottom.

When viewing the reconstructed images after compression using lossy techniques, the rms value is somewhat subjective with respect to the quality of the original. In this report, arbitrary rms limits were assigned based on parameters set by the Television Allocations Study Organization (Gonzalez, 1987, pp. 257-258).

* Excellent - An image of extremely high quality, as good as you could desire ($0 \leq e_{rms} < 6$).

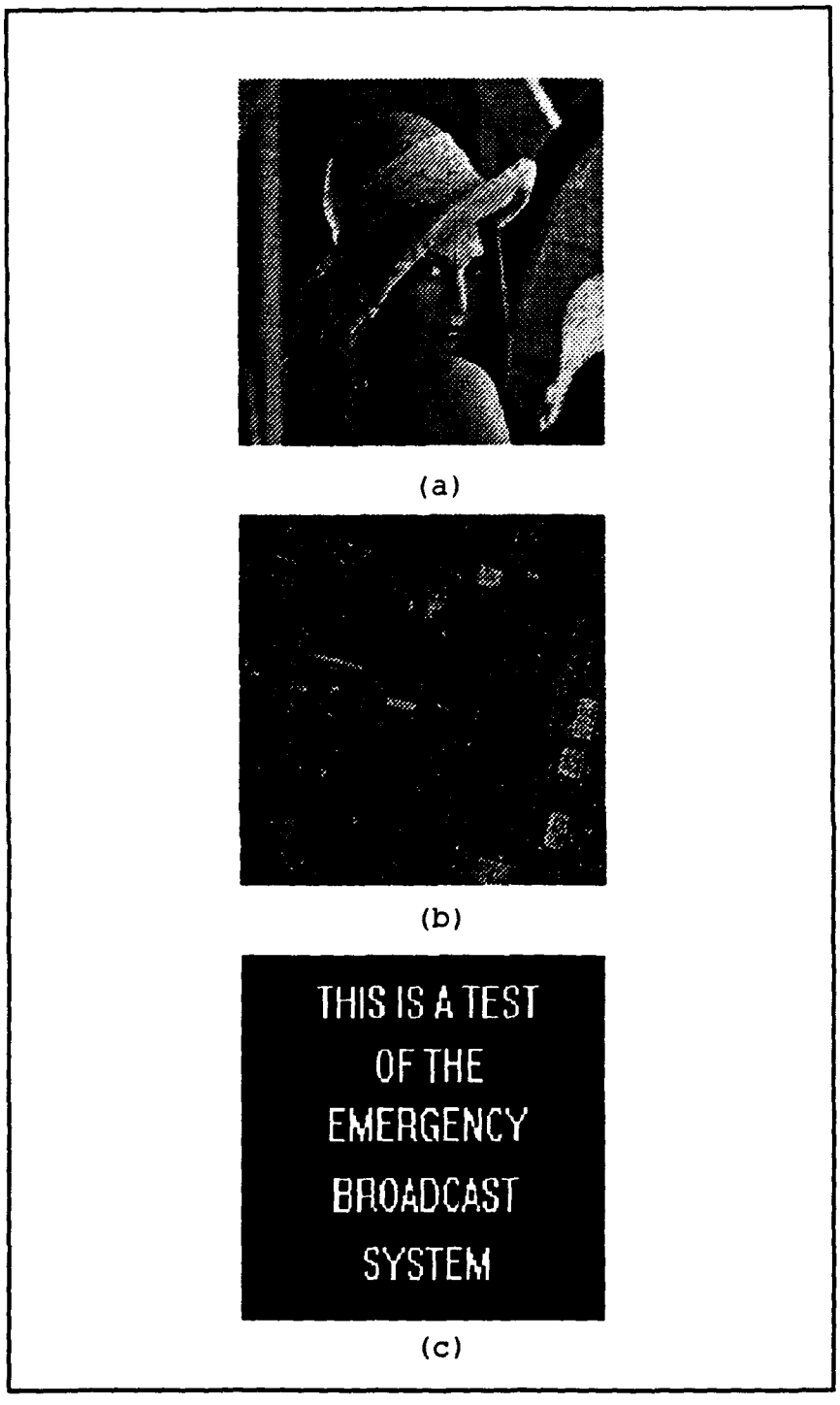

Figure IV.1: Original images (a) LENA, (b) AEIAL (c) SIGN.

* Fine - An image of quality, providing enjoyable viewing. Interference is not objectionable ($6 \leq e_{rms} < 9$).

* Passable - An image of acceptable quality. Interference is not objectionable ($9 \leq e_{rms} < 12$).

* Marginal - An image of poor quality; you wish you could improve it. Interference is somewhat objectionable ($12 \leq e_{rms} < 14$).

* Inferior - A very poor image, but you could watch it. Objectionable interference is definitely present ($14 \leq e_{rms} < 17$).

* Unusable - An image so bad that you could not watch it ($17 \leq e_{rms}$).

In order to provide the reader with an idea as to how this can affect image quality, Figure IV.2 shows LENA for an increasing rms.

With the JPEG, rms is controlled by the input of a variable called quality factor, which is an integer from 1 to 25. The quality factor is directly related to the quantization matrix. The upper left-hand value becomes the quality factor plus one, and every variable in the next diagonal is increased by the quality factor. This increase continues through each diagonal. See Figure III.4 for an example of the quantization block for an input quality factor of two. A rising quality factor causes a corresponding increase in rms, consequently the decompressed image fidelity deteriorates.

The rms for Fractal compression is varied by a user input called the error cut. The error cut is the same as the least-mean-squares value defined in formula III.5. An additional

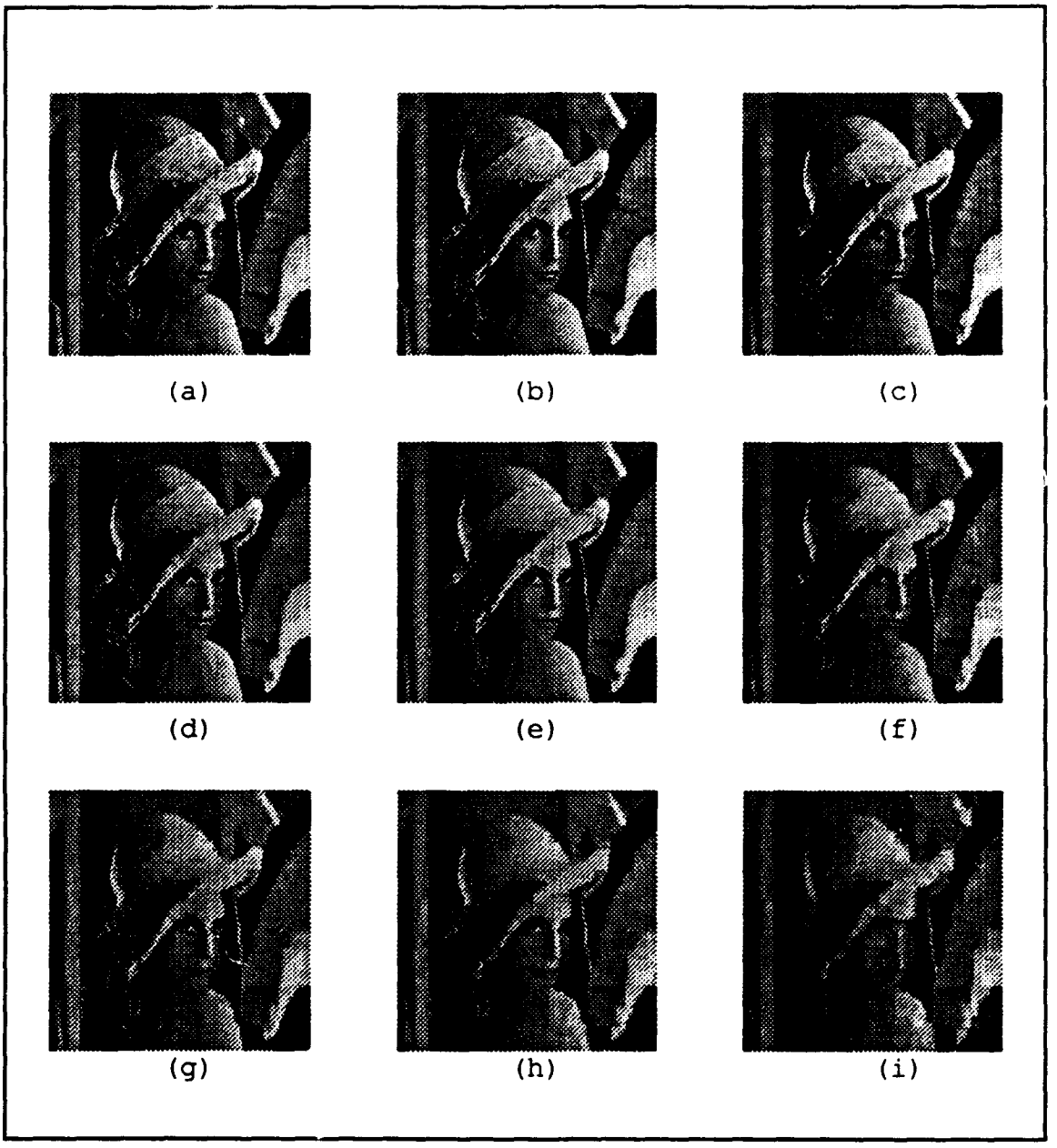

Figure IV.2: LENA for varying rms values (a) original, (b) 4.57 (c) 7.04 (d) 9.28 (e) 11.07 (f) 12.99 (g) 14.31 (h) 16.47 (i) 19.05.

user input called optimization level is also required. This value varies the possible domain shape, size, and location, in addition to range/domain classifications; thus speed and compression capability can be modified.

Figures IV.3 and IV.4 show graphically how rms changes with varying quality factor inputs for JPEG and error cut inputs for Fractal compression. For high quality images (low quality factor and error cut values), a change in the quality factor or error cut has a much more noticeable effect on rms. As image quality degrades though, changes in the two variables have less of an impact on the resulting rms. One exception is the Fractal compression of SIGN. At a certain point, the routine is unable to improve rms. Later graphs will also show that the compression ratio reaches a limit as well.

Figure IV.5 is a comparison of rms versus compression ratio using the JPEG algorithm. For the most part, there is a linear relationship between the two properties. Looking at Figure IV.6, at least for processing of LENA and AERIAL, this is not the case with Fractal compression. There is a super-linear relationship at the very least. At high quality reconstruction, a small increase in compression can cause a large drop in fidelity. On the other hand, at inferior qualities, compression may be drastically increased for only a small loss in fidelity. Again, SIGN is the exception because there is a linear relationship between the two properties until a point

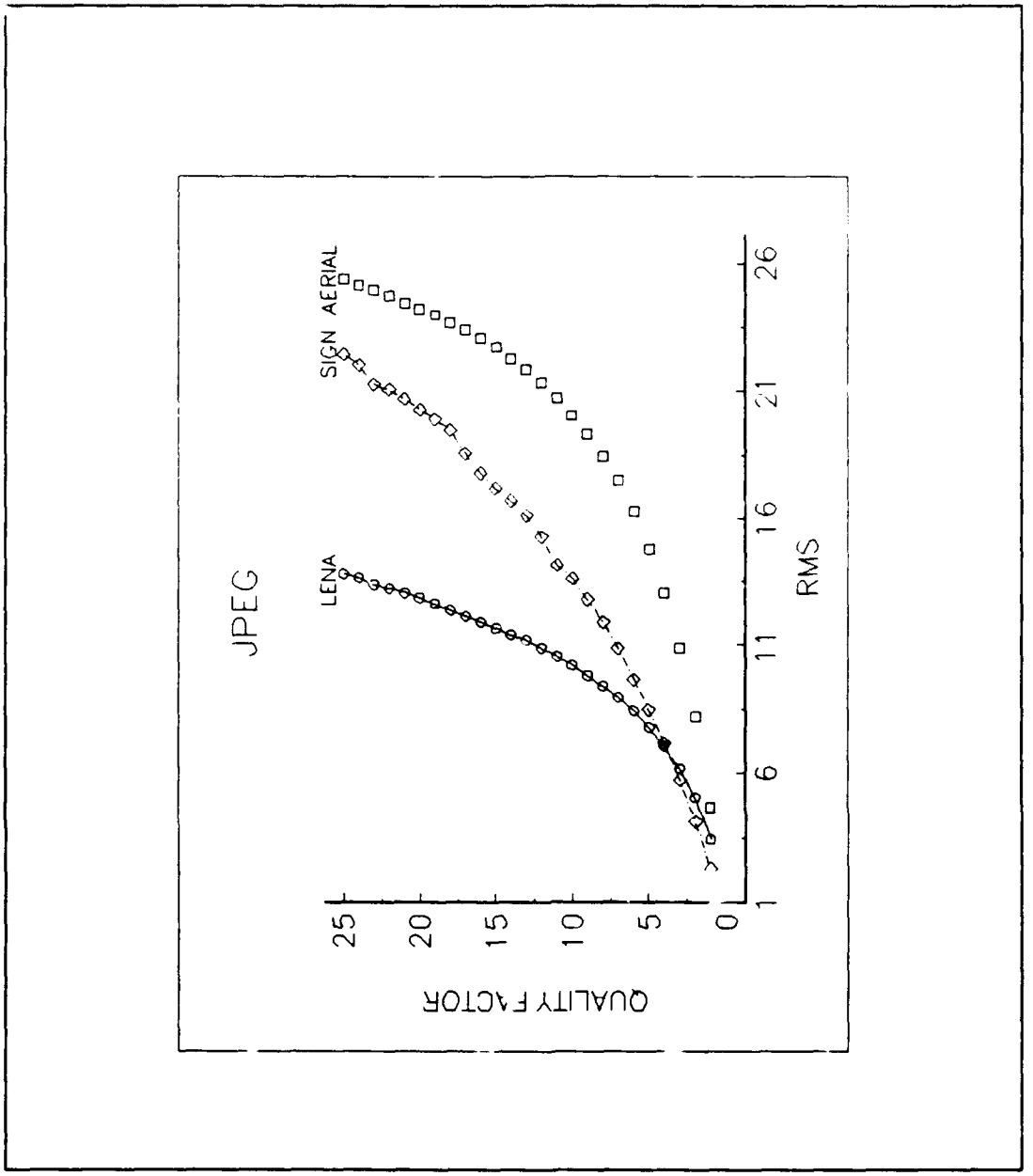

Figure IV.3: Comparison of root-mean-square to quality factor for JPEG compression.

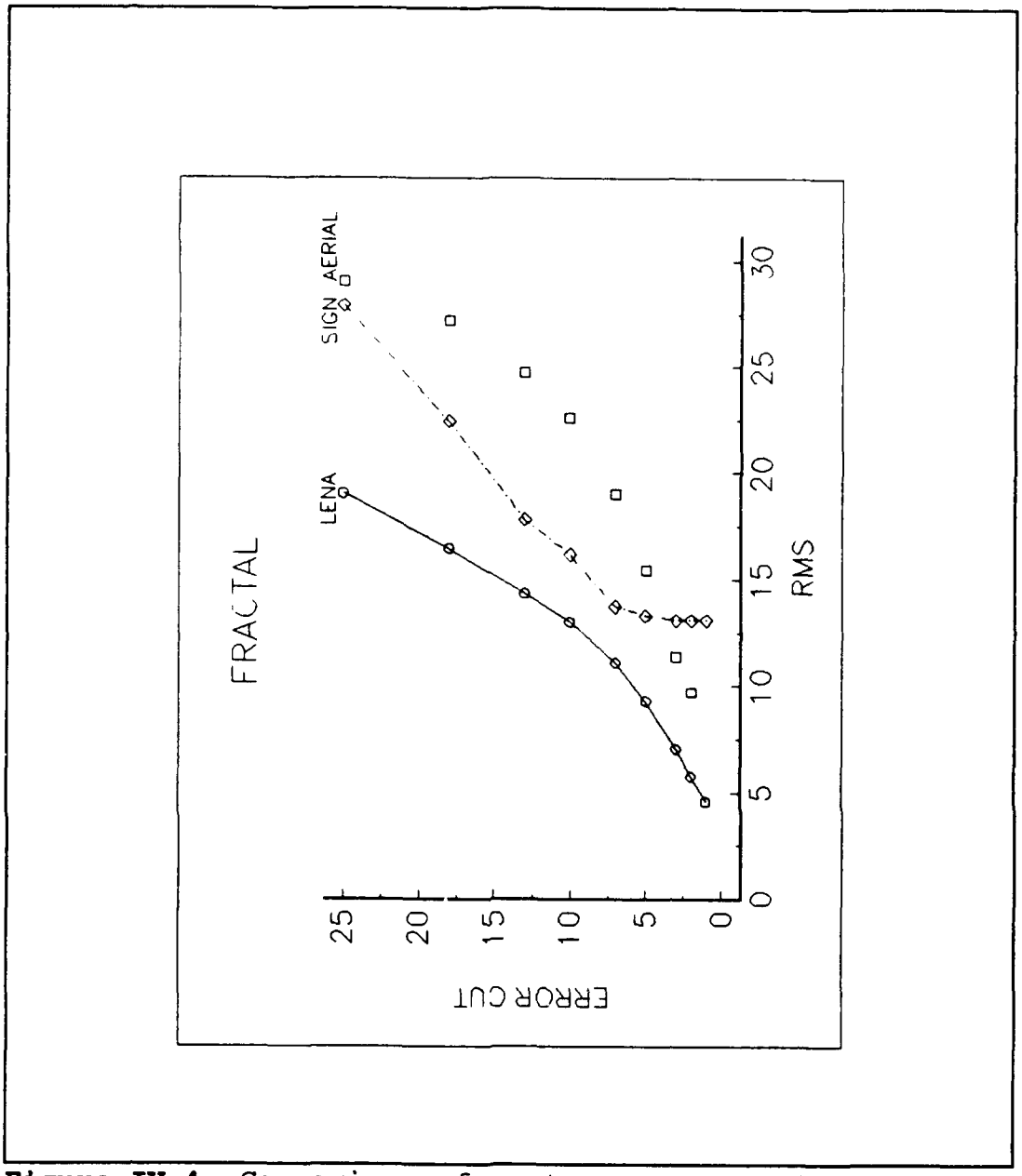

Figure IV.4: Comparison of root-mean-square to error cut for Fractal compression.

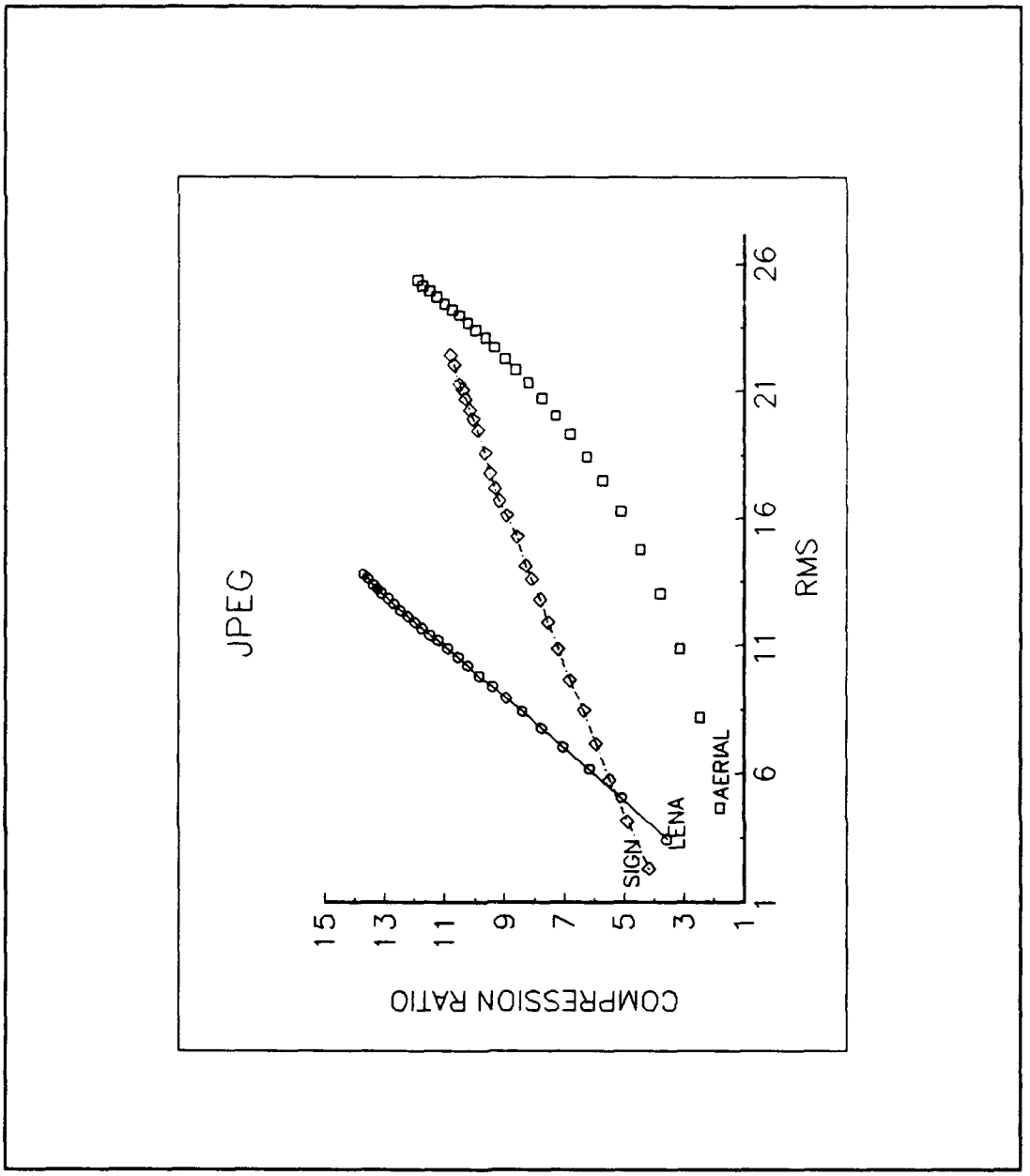

Figure IV.5: Comparison of root-mean-square to compression ratio for JPEG compression.

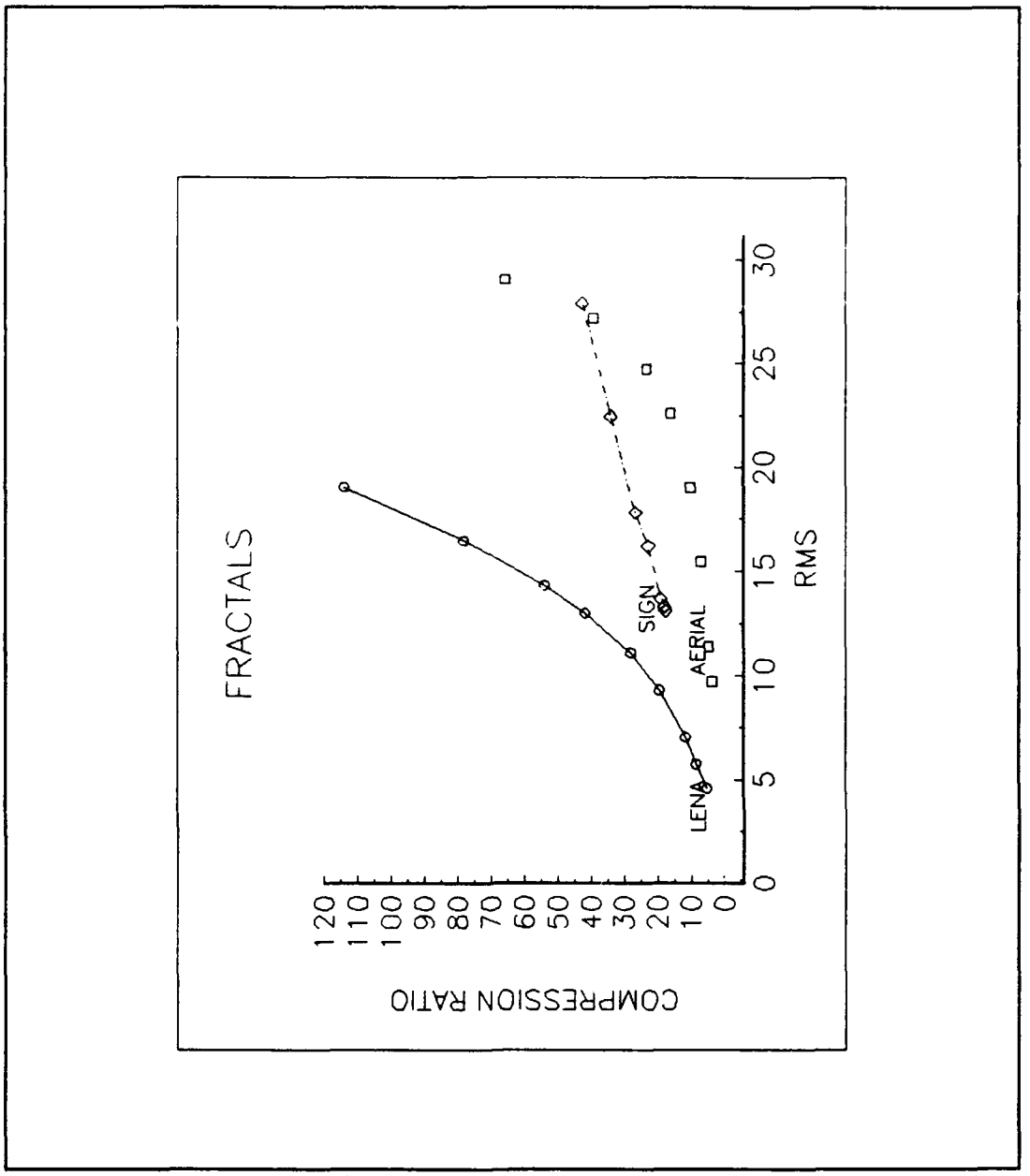

Figure IV.6: Comparison of root-mean-square to compression ratio for Fractal compression.

where compression reaches a limit. Additionally, the rms never gets better than unusable. In actuality, the image is discernible, but there is a great deal of distortion or loss of distinct boundary between the white letters and black background; thus its use for commercial applications in this instance is questionable. This boundary ripple is also evident with JPEG compression/decompression, even at low rms.

Figure IV.7 demonstrates when using JPEG compression at high fidelity, significant savings can be achieved in compression/decompression time with small drops in image reconstruction quality. The reverse is true at poor quality - the compression/decompression time reaches a point where reducing the fidelity does not result in any savings in time, thus the only benefit is decreased data storage requirements.

A comparison of the compression/decompression time versus the compression ratio for each method, applied to all three sample images, can be seen in Figure IV.8. The results are based on the best obtained rms in the case of the JPEG and Fractal routines. It should be noted that since the Huffman, Adaptive Huffman, and Arithmetic compression methods are lossless, the rms of the decompressed image is zero in each case. Due to vast differences in time for the Fractal routines, compression/decompression time is represented by taking its log base 10. For LENA and AERIAL, the Fractal technique achieves greater compression, but at the expense of

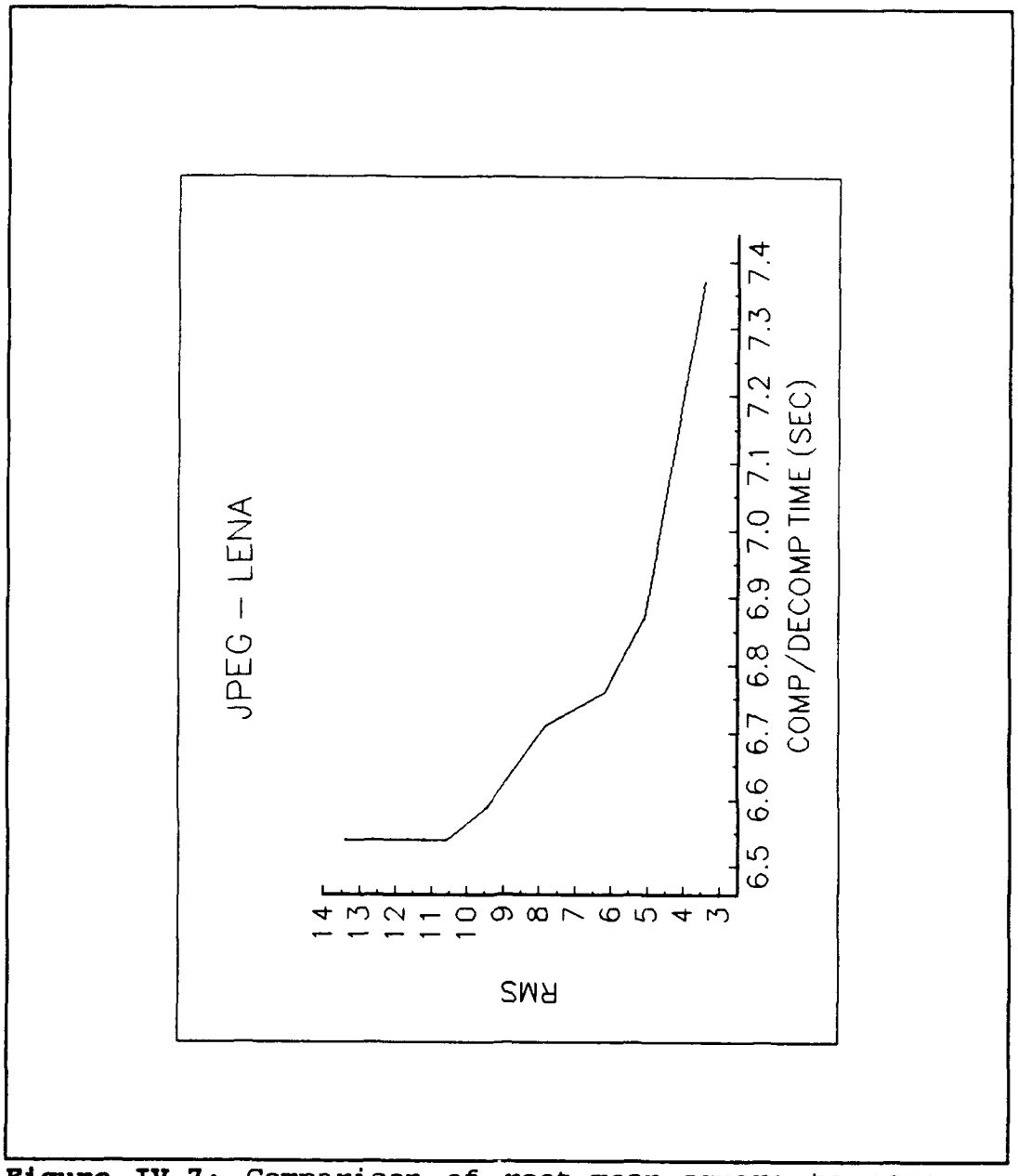

Figure IV.7: Comparison of root-mean-square to compression/decompression time for JPEG compression of LENA.

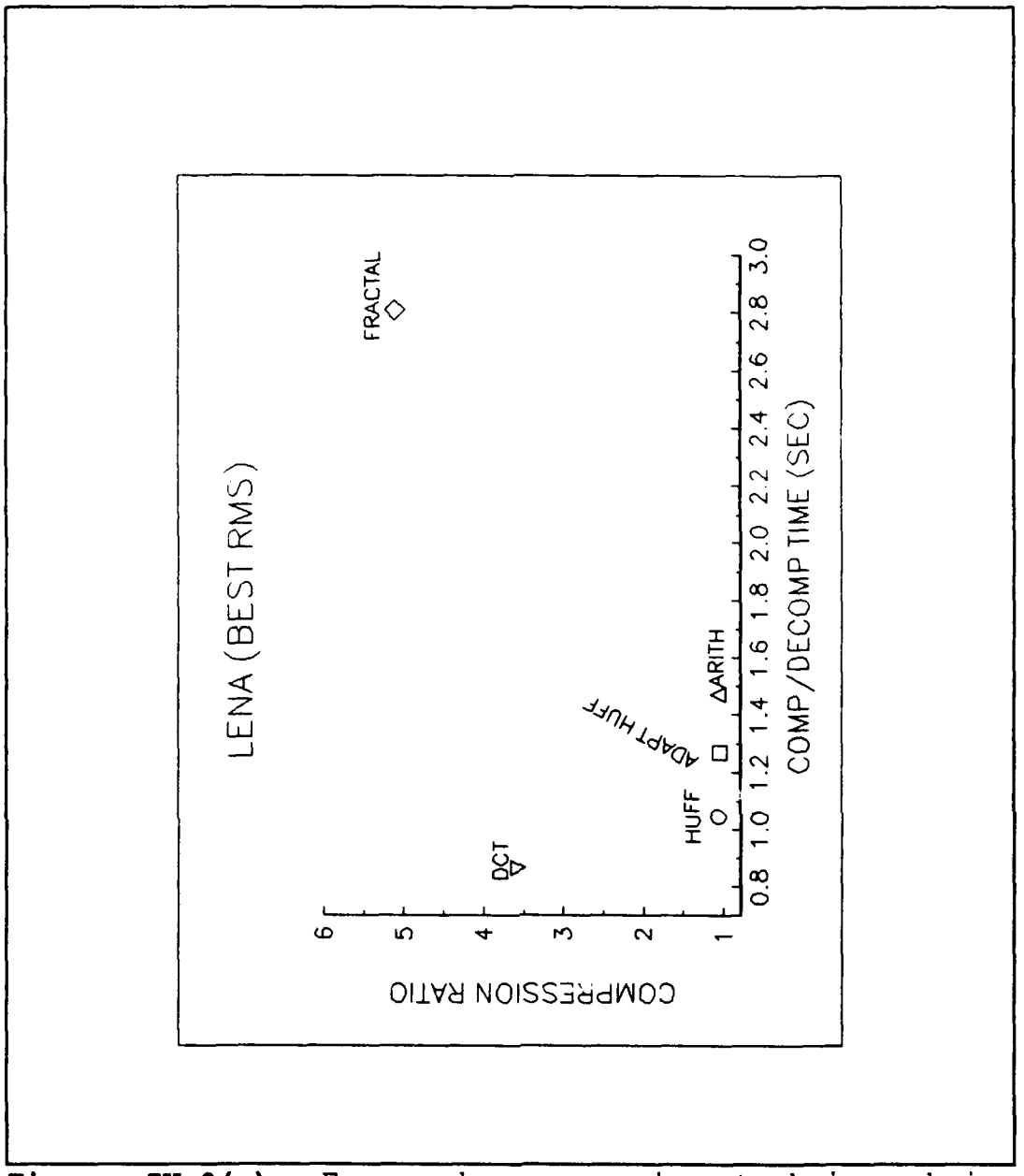

Figure IV.8(a): For each compression technique being performed on LENA, a comparison of compression/decompression time to compression ratio.

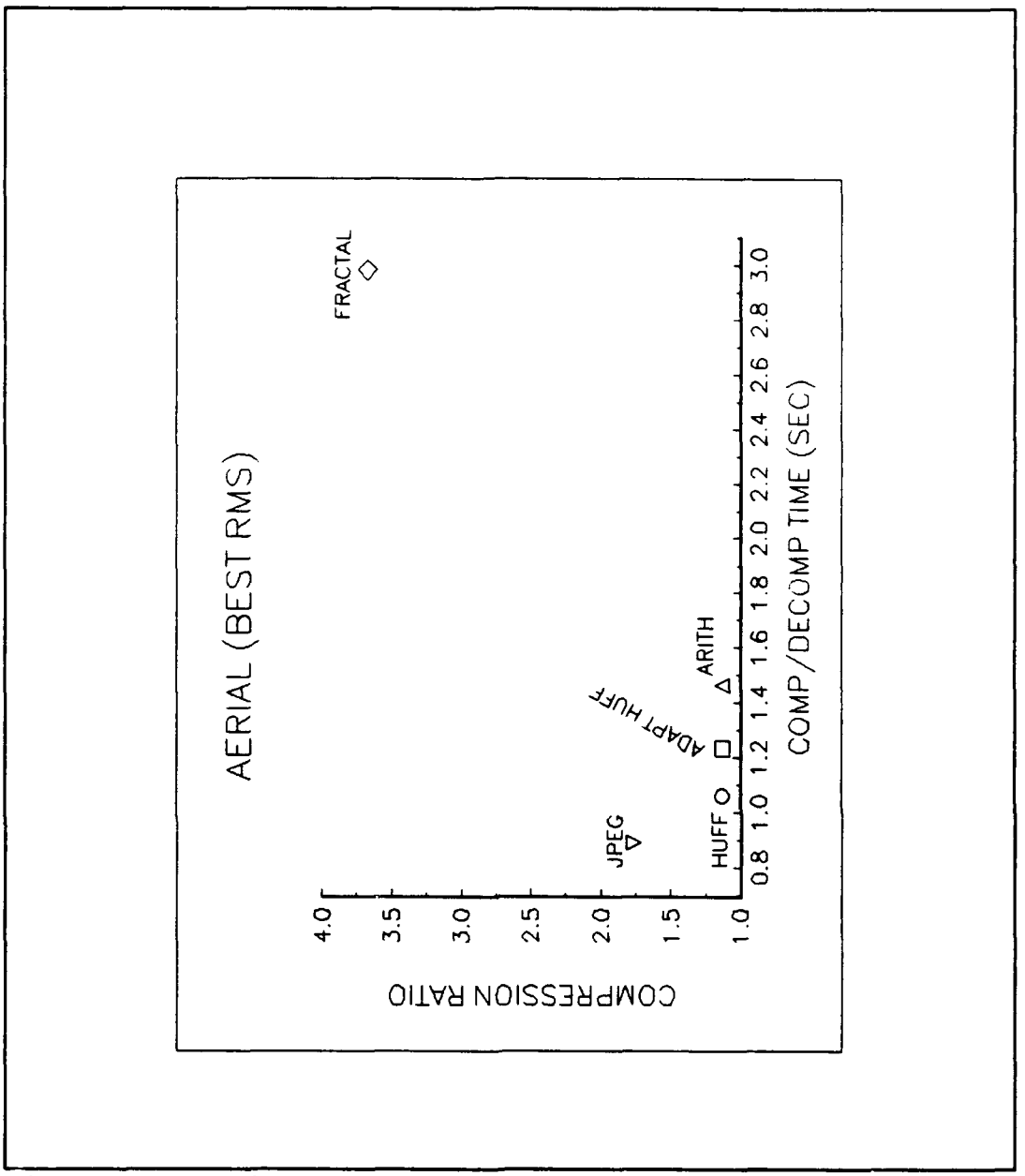

Figure IV.8(b): For each compression technique being performed on **AERIAL**, a comparison of compression/decompression time to compression ratio.

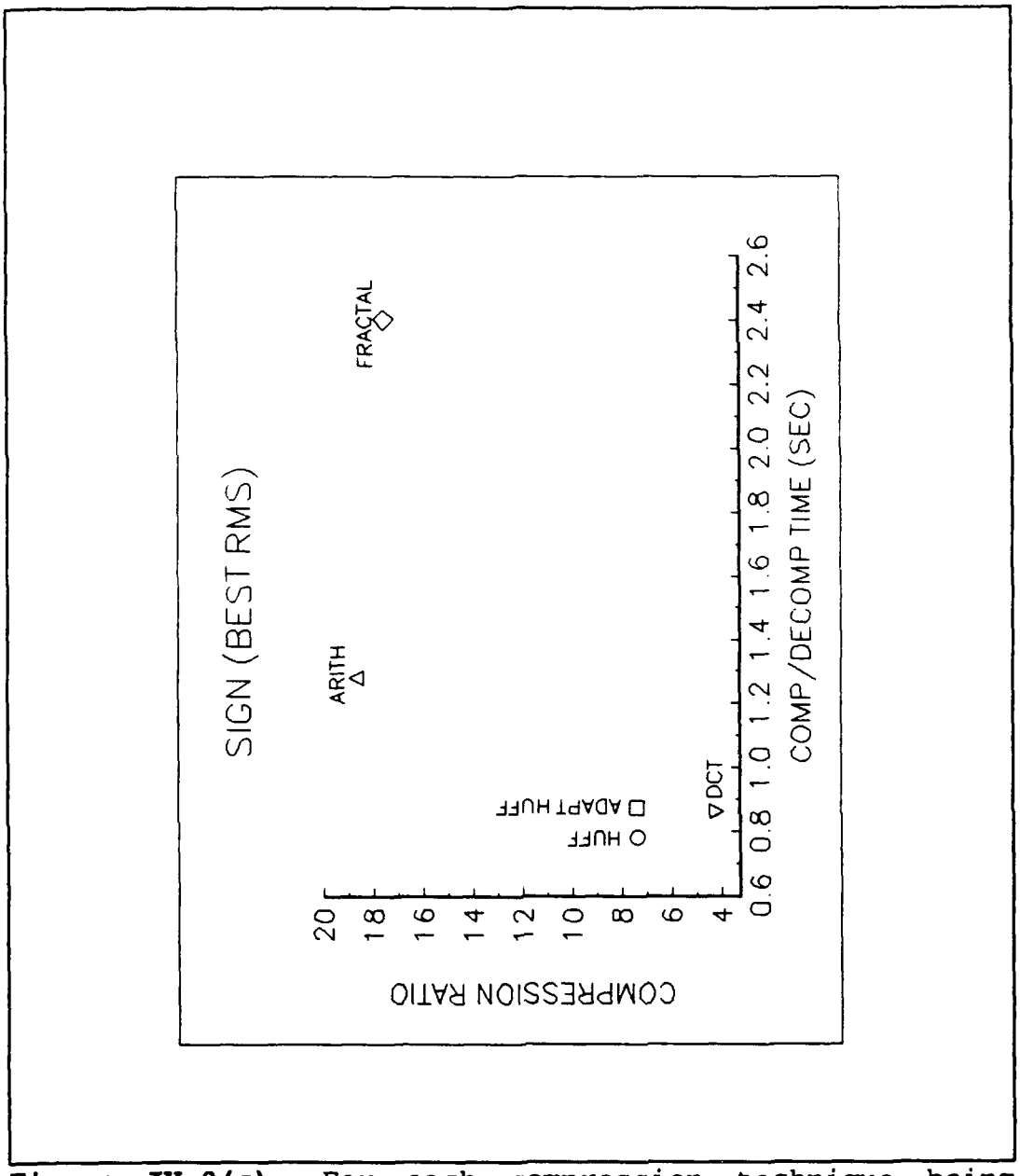

Figure IV.8(c): For each compression technique being performed on SIGN, a comparison of compression/decompression time to compression ratio.

a large increase in time. In the case of SIGN, the arithmetic and Fractal methods are comparable in compression, with arithmetic coding being somewhat better. Arithmetic coding though, is able to achieve the reduction in much less time. Moreover, its decompressed image does not have the ripple at the boundary between the black and white colors. One additional note refers to Chapter II.C, which states that black and white (one bit per pixel) images could not be compressed using Huffman coding. In the case of SIGN however, compression is in fact achieved. The reason is that in this case the two colors are each represented by eight-bits, not one bit, hence a reduction in the data-bit representation is possible. The numerical data results for the figures are contained in Appendix C.

V. CONCLUSIONS

An analysis of the results from the previous chapter indicates that for multi-color, real world images, the JPEG compression algorithm is currently the best method for image compression. Though the Fractal method is able to achieve a much higher compression ratio for similar rms values, its present execution time and computing power requirements (Gershanoff, 1988, p. 47), (Barnsley, 1988, p. 222) severely limit its practicality for present applications. In its defense, the theory and principles behind Fractal image compression are still relatively young. As new methods are discovered to classify and partition the domain set and find the affine transformations needed to map the domain to the attractor (range), this technique will only improve. The continual advancement in computer processing capability will also assist. For these reasons, Fractal compression offers the most potential for future applications, such as high definition television (HDTV) and image recognition for satellite imagery.

Another advantage of Fractal over JPEG compression is that image size can be changed during decompression since any scaling is multiplied proportionally through the affine transformation at each iteration (Anson, 1991, p. 43). The

JPEG is further limited in the image size it can process. The 8x8 block division criterion reduces possible dimensions to those divisible by eight on each side. This 8x8 rule can be modified, but then the model does not meet JPEG standards.

In the case of SIGN - a highly redundant, two color image - the lossless method of Arithmetic coding is easily the preferred technique. The JPEG and Fractal algorithms are not very effective because even at high quality, there is a noticeable ripple introduced at the boundary between the white letters and black background. Huffman and Adaptive Huffman coding are also better methods, considering the compression/decompression time, compression obtained, and decompressed image resolution. Therefore, when compressing two-color images (black and white), lossless algorithms are certainly the favored procedure since the introduction of errors can seriously flaw the decompressed image.

Future areas of research might introduce other proven image compression techniques into the comparison. These include higher order Arithmetic coding and the Ziv and Lempel (LZ78) dictionary based schemes. The speed-up of the Fractal compression routine by altering range/domain classifications and domain partitioning, or modifying the methods used to find the affine transformations, is another prospect. A third concept is the optimization of image compression with a combination of two or more of the possible algorithms. The

JPEG method, which combines quantization with lossless entropy compression, is an example of a two part compression process. With the continuing demand for image data though, more efficient methods of compression will need to be discovered, and the combining of more than one technique is a distinct possibility.

APPENDIX A - OPERATIONAL MECHANICS

A. CONCERNS

The inclusion of this appendix is to assist anyone who may desire to continue research in the area of image compression. The difficulties experienced in getting the compression algorithms to run will be discussed. It is anticipated that the reader, if deciding to explore this subject area further, will obtain a fundamental understanding of the operational mechanics and thus, avoid the steep learning curve usually associated with new software.

B. SPECIFIC DIFFICULTIES

A detailed list of the specific hurdles encountered during the research and how each one was approached.

1. Image Format

The first area of difficulty deals with image format. There are a wealth of possibilities. The documentation provided with the *Graphics Transformer* (IMSI, 1990) is a good reference for explaining the differences between various format types. For this work, as stated earlier, the format is raw pixel grey-map (*.rpg, *.rpgm, or *.raw). In truth, this is the closest thing possible to a "non-format". Data is listed from left to right, row by row. Each ASCII character

represents one of 256 shades of gray. While working with the images though, one will likely become familiar with many additional format types. Raster (*.ras) is utilized for working in a Sun System image processing tool called Sun-vision. This format is structured with controlling data placed throughout the image data. Each well-known word processing package utilizes its own format - (*.wpg) for WordPerfect, (*.pic) for Lotus 1-2-3, (*.pcx) for PC Paint-brush, (*.dxf) for AutoCAD, etc. Many of the packages are able to import files stored in another format, but its reference should be checked if unsure.

2. Sun System to PC & PC to Sun System

If the image data is not in the required format for processing, the Sun System provides a conversion tool called *imconv*. It will convert approximately 30 different image format types. It can be accessed by setting a path to */tools2/imagecv/bin/*. To see the reference for its execution, type *imconv -fullhelp* at the cursor. Also available for format conversion, are numerous PC software packages. One utilized by this author is *Graphics Transformer* (IMSI, 1990). *Imconv* was faster though, and offered many more options than its PC counterpart. The image files are transferred from floppy disc to Sun account, and vice versa, with *mtools*. Documentation can be obtained in Sp-301.

3. Display

There are several alternatives for image display. The Sun System in the Electrical and Computer Engineering (ECE) Department offers Sunvision. It can be found in the path *tools2/sunvision_1.1/bin/*. Reference documentation is maintained in the ECE image Processing Lab in Sp-546. This software is somewhat outdated, thus there are obstacles to getting it operational under the current Sun System Windows Version 3 (OW3). It can be used only in Open Windows Version 2 (OW2). The best advice is to ask the account manager in Sp-301 to provide two personal accounts, one that logs in under *OW2*, and another which logs in under *OW3*. In this fashion, Sunvision is accessible and the upgraded capabilities of the newer windows is still available. While in account #1, a file can be moved from account #1 to account #2 by using the *cp* command. First transfer it to the */temp* directory, change its accessibility to read/write using the *chmod* command, and after a remote log in to account #2, move the file from */temp* to account #2. Be aware that all Sun terminals are not capable of running *OW2*. A second Sun option is to write a display program. For a beginning C programmer, this would not be overly difficult since the Sun graphics are fairly user friendly.

Another graphics tool is the *xv* editor on the Silicon Graphics machines in the Visualization Lab, In-148. Reference

documentation can be acquired by typing *man xv* at the terminal cursor. Besides display, both options offer scaling, rotation, color altering, and more. This provides the flexibility to change the image for varying research goals.

Display is more difficult on the PC because there are a vast assortment of video card drivers, a component needed for graphics display. Additionally, PC monitors are not as capable as Sun monitors; thus a C language display program is much more entailed than that needed for Sun monitor display. One suggestion is to check if the supplier of the image processing software has a display program available. For this study, Netrologics (1993) provided a display program called *rawview.exe*.

4. Printing

An image printout can be obtained on the Sun by first converting the file into postscript (*.ps) format and then typing *lpr<printer#> <filename>*. Another option is to import the image into a word processing package. Of course, the file will need to be in a format that is readable to the package utilized. In terms of size and page layout, this author found FrameMaker (available on the Sun) to be very flexible with imported files. It requires them to be in Raster format.

5. Access

a. Fractal Program

The Fractal compression/decompression programs in Appendix B can be found under the public access address *lyapunov.ucsd.edu*. Transfer the files to a personal Sun account using the *ftp* command. Type *man ftp* at the prompt for instructions on the use of this command. After gaining access to the public access network, type in *anonymous* when asked for user name, and personal e-mail address when asked for password. The programs are listed in the directory */pub/youngfractal* as *unifs10.tar.Z*. This is a compressed archive file. To get the individual files, become familiar with the *tar* and *compress* commands (type *man tar* and *man compress* at the prompt).

b. Nelson's Compression Routines

The computer programs listed in Nelson's book can be purchased by calling the publisher (which recently changed to Henry Holt Publishing) at 1-800-488-5233.

c. Compilers

The compiler for the Appendix B Fractal programs can be obtained on the public access network at *omnigate.clarkson.edu*, which can be accessed in the same fashion described earlier. The directory is *pub/msdos/djgpp*. The required files are *djdev109.zip*, *djgas138.zip*, *djgcc222.zip*, *djlgr110.zip*, and *readme.1st*. The *readme.1st* and *readme* files

provide all the instructions for installing the compiler onto a PC. One thing not mentioned is that the *libgr.a* file in *djlgr109.zip* needs to be added to the library directory once the compiler is installed.

Nelson's programs can be compiled using Borland C++ 2.0 for MS-DOS (1992, pp. 5-6, 78-79). Since version 2.0 is obsolete, version 3.0 was found to compile with one exception to the example command line on page 79. The -Ax option no longer exists, the replacement is just -A. The programs will not compile under the Borland MS-WINDOWS version. They will also not compile using the *cc* compiler on the Sun System. This requires such significant program modifications that they must practically be rewritten.

d. Utilities

The account manager must set up a Sun account for access to FrameMaker, Sunvision, etc. Be sure to specify the utilities needed when signing up for an account.

PC utilities (Borland C++ compiler, PaintBrush, WordPerfect, DrawPerfect, etc.) are installed on the computers in Sp-431. Reference documentation, if not available, can usually be obtained from the lab technician.

6. Compiling and Running

a. Fractal Routine on a PC

Upon installing the compiler as per paragraph 5.c above, the program fracpack.c for instance, can be compiled

using the command line *gcc fracpack.c -lgr -lm -o fracpack*. The order of the library options (*-lgr* and *-lm*) is important because the linking is order-sensitive. The program *wdy-usual.c* in appendix B must be in the same directory as *fracpack.c* because it is called to classify the domains and ranges. The compiled file must then be appended to the file *go32.exe* (a program which provides graphics driver interface) using a program called *aout2exe.exe*. Both files are contained in the compressed *unifs10.tar.Z* file. Typing *aout2exe fracpack* will create a file that is PC executable.

Since the programs *fracpack.c* and *unifs.c* display the image during execution, the graphics driver used by *go32.exe* must be set by the user. Directions are listed in the *readme* and *document* (*.doc) files, which are also included in *unifs10.tar.Z*. If the proper driver is not set, the computer will lockup.

b. Fractal Routine on the Sun System

As they are listed in Appendix B, the Fractal compression/decompression programs will not run on the Sun System. The graphics are not compatible. The programs can be modified by removing any code dealing with the display of the file. The programs can then be made Sun executable by compiling with the *cc* compiler (type *man cc*). In this case, appending to the program *go32.exe* is not required since PC monitor display is not being used.

c. Nelson's Programs on a PC

With the proper compiler, Nelsons's programs run with no problems. The user should be informed that the programs were written for images of 320x200 pixels (1992, pp. 374), therefore some modifications might be necessary in the code in order to process images with different dimensions.

APPENDIX B

This is the file "copying.wy".

Copyright Information for sources and executables that are marked Copyright (C) WD Young
P.O. Box 632871
Nacogdoches TX 75963-2871

This document is Copyright (C) WD Young and may be distribute verbatim, but changing it is not allowed.

Source code copyright WD Young is distributed under the terms of the GNU Public License, with the following exceptions:

*Donations are always appreciated.

A copy of the file "COPYING" is included with this document. If you did not receive a copy of "COPYING", you may obtain one from whence this document was obtained, or by writing:

Free Software Foundation
675 Mass Ave
Cambridge, MA 02139
USA

```
/*****************************************************************************
                        FRACPACK - a program for
                        FRACTAL IMAGE COMPRESSION
                         image.krd ==> image.ifs
                              version 1.0
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
#include <math.h>
#include <time.h>
#define xsize 256
#define ysize 256
#define number_flips 8
#define levels 2
#define max_patch 8
#define min_patch 4
#define max_scale 1.2
long int usual (unsigned char Image[8][8], int size);
int mapping;
int main(int argc, char **argv)
{  /* begin main block */
    FILE *in, *out, *cf;
    char *inf, *outf, rmsstr[13];
    unsigned char Image[ysize][xsize], blur[ysize-1][xsize-1], plotdx, plotdy;
    unsigned char Range[number_flips][max_patch][max_patch], Domain[max_patch][m
    unsigned char DX[xsize*ysize], DY[xsize*ysize];
    int reverse;
    long int Domain_Class[levels][ysize - min_patch + 1][xsize - min_patch + 1][
    long int Range_Class[levels][max_patch][max_patch][2];
    int i, rx, ry, dx, dy, x, y, besti, bestdx, bestdy, patchsize;
    long int class1, class2, class3, class4, count;
    int inlevel;
    int I[8][8] = {{0,3,2,1,4,5,6,7},{1,0,3,2,7,4,5,6},{2,1,0,3,6,7,4,5},{3,2,1,
                   {4,7,6,5,0,1,2,3},{5,4,7,6,3,0,1,2},{6,5,4,7,2,3,0,1},{7,6,5,4
    short int offset, best_offset;
    long int sumr, sumd, sumrd, sumr_sq, sumd_sq;
    long int Domain_sums[2][xsize - min_patch + 1][ysize - min_patch + 1];
    float fsumr, fsumd, fsumrd, fsumr_sq, fsumd_sq, fmagica;
    float best_scale, scale, root_mean_sq, best_root_mean_sq;
    float mean_sq, root_mean_sq_tolerance, mean_sq_tolerance, best_mean_sq, fpat
    float temp, variance;
    time_t start_time, finish_time;
    long time_used;

    struct header_t {    /* "should" be a 12 byte header... we'll see */
      long time;         /* 4 bytes for compression time in seconds */
      short rms;         /* 2 bytes for 100.*rms value */
      short add1;        /* 2 bytes to be added later... room for growth */
      long add2;         /* 4 bytes to be added later... room for growth */
      } header[1];

    struct ifs_t {
      unsigned char dx;
      unsigned char dy;
      signed char scale;
      short int offset : 12;
      unsigned short int flip : 3;
      unsigned short int size : 1;
      } ifs[1],
```

```
        ifs_table[levels][xsize/min_patch][ysize/min_patch];

    if (argc < 4)   {
        printf("usage: fracpack rms infile.ext outfile.ext \n\n");
        printf("FRACPACK Version 1.0, Copyright (C) 1992, WD Young\n");
        printf("FRACPACK comes with ABSOLUTELY NO WARRANTY\n");
        printf("Please see files 'copying.wy' and 'copying' for details\n");
        printf("If these files are missing,\n");
        printf("write: WD Young, P.O. Box 632871, Nacogdoches TX 75963-2871\n");
        return 1;
        }


    root_mean_sq_tolerance = atof(argv[1]);
    header[0].rms = (short)(100.*root_mean_sq_tolerance);

    inf = argv[2]; outf = argv[3];
    if ((in = fopen(inf, "rb")) == NULL)    {
        fprintf(stderr, "Cannot open input file.\n");
        return 1;
        }
    if ((out = fopen(outf, "wb")) == NULL)  {
        fprintf(stderr, "Cannot open output file.\n");
        return 1;
        }
    fclose(out);

    start_time = time(start_time);
    mean_sq_tolerance = root_mean_sq_tolerance*root_mean_sq_tolerance;

    GrSetMode(GR_default_graphics);
    for (x = 0; x < 64; x++)
        GrSetColor(x,4*x,4*x,4*x);
    for (x = 64;x < 256; x++)
        GrSetColor(x,x,0,0);


/* Get .KRD bitmap 8-bit-grey-scale pixels, put in Image array.  */
    for (y = 0; y < ysize; y++)
    for (x = 0; x < xsize; x++)  {
        Image[y][x] = fgetc(in);
        GrPlot(x,y,Image[y][x]>>2);
        GrPlot(x+300,y,Image[y][x]>>2);
        }

    for (y = 0; y < ysize - 1; y++)
    for (x = 0; x < xsize - 1; x++)
        blur[y][x] =  (Image[y  ][x  ]
                     + Image[y  ][x+1]
                     + Image[y+1][x  ]
                     + Image[y+1][x+1])>>2;

    fclose(in);
    sprintf(rmsstr,"%4.1f",sqrt((double)mean_sq_tolerance));
    GrTextXY(20,290, "rms tolerance", 255, 0);
    GrTextXY(130,290,rmsstr,255,0);


/*************************************************************************/
/*                          Classify Range's                             */
```

```
/*************************************************************************/
    inlevel = 0;
    for (ry = 0; ry < ysize; ry+=8) {
        GrLine(0, ry, 255, ry, 384);
        for (rx = 0; rx < xsize; rx+=8)
            {
            for (y = 0; y < 8; y++)
            for (x = 0; x < 8; x++)
                Range[0][y][x] = Image [ ry  +y ] [ rx  +x ];
            class1 = usual(Range[0], 8);
            Range_Class[inlevel][ry>>3][rx>>3][0] = class1;
            Range_Class[inlevel][ry>>3][rx>>3][1] = mapping;
            }
        GrLine(0, ry, 255, ry, 384);
        }


/*
*****************************************************************
*                                                               *
*  Compute Domain_sums array                                    *
*                                                               *
*****************************************************************
*/
    inlevel = 1;
    patchsize = min_patch;
    for (dy = 0; dy < ysize - 2*patchsize+1; dy++) {
        GrLine(300, dy, 555, dy, 384);
        for (dx = 0; dx < xsize - 2*patchsize+1; dx++)  {
            Domain_sums[0][dy][dx] = Domain_sums[1][dy][dx] = 0;
            for (y = 0; y < 2*patchsize; y+=2)
            for (x = 0; x < 2*patchsize; x+=2)  {
                Domain[y>>1][x>>1] = blur[dy+y][dx+x];
                Domain_sums[0][dy][dx] += Domain[y>>1][x>>1];
                Domain_sums[1][dy][dx] += Domain[y>>1][x>>1]*Domain[y>>1][x>>1];
                }
            variance = (Domain_sums[1][dy][dx]- Domain_sums[0][dy][dx]*Domain_su
            if (variance > 16) {
                class1 = usual(Domain, 4);
                Domain_Class[inlevel][dy][dx][0][0] = class1;
                Domain_Class[inlevel][dy][dx][0][1] = mapping;

                for (y = 0; y < 2*patchsize; y+=2)
                for (x = 0; x < 2*patchsize; x+=2)
                    Domain[y>>1][x>>1] = 255 - blur[dy+y][dx+x];
                class1 = usual(Domain, 4);
                Domain_Class[inlevel][dy][dx][1][0] = class1;
                Domain_Class[inlevel][dy][dx][1][1] = mapping;
                }
            else Domain_Class[inlevel][dy][dx][0][0] =
                 Domain_Class[inlevel][dy][dx][1][0] =-1;
            }
        GrLine(300, dy, 555, dy, 384);
        }

/*************************************************************************/
/*                          Classify Range's                             */
/*************************************************************************/
    for (ry = 0; ry < ysize; ry+=4) {
        GrLine(0, ry, 255, ry, 384);
```

```
        for (rx = 0; rx < xsize; rx+=4)
            {
            for (y = 0; y < 4; y++)
            for (x = 0; x < 4; x++)
                Range[0][y][x] = Image [ ry  +y ] [ rx  +x ];

            class1 = usual(Range[0], 4);
            Range_Class[inlevel][ry>>2][rx>>2][0] = class1;
            Range_Class[inlevel][ry>>2][rx>>2][1] = mapping;
            }
         GrLine(0, ry, 255, ry, 384);
        }
/*
*****************************************************************
*                                                               *
*  Compute Domain_sums array                                    *
*                                                               *
*****************************************************************
*/
    patchsize = max_patch;
    inlevel = 0;
    for (dy = 0; dy < ysize - 2*patchsize+1; dy++) {
        GrLine(300, dy, 555, dy, 384);
        for (dx = 0; dx < xsize - 2*patchsize+1; dx++)  {
            sumd = sumd_sq = 0;
            for (y = 0; y < 2*patchsize; y+=2*min_patch)
            for (x = 0; x < 2*patchsize; x+=2*min_patch) {
                sumd += Domain_sums[0][dy + y][dx + x];
                sumd_sq += Domain_sums[1][dy +  y][dx + x];
                }
            variance = (sumd_sq - sumd*sumd/63.)/64.;
            if (variance > 16) {
                for (y = 0; y < 2*patchsize; y+=2)
                for (x = 0; x < 2*patchsize; x+=2)
                    Domain[y>>1][x>>1] = blur[dy+y][dx+x];


                class1 = usual(Domain, 8);
                Domain_Class[inlevel][dy][dx][0][0] = class1;
                Domain_Class[inlevel][dy][dx][0][1] = mapping;

                for (y = 0; y < 2*patchsize; y+=2)
                for (x = 0; x < 2*patchsize; x+=2)
                    Domain[y>>1][x>>1] = 255 - blur[dy+y][dx+x];

                class1 = usual(Domain, 8);
                Domain_Class[inlevel][dy][dx][1][0] = class1;
                Domain_Class[inlevel][dy][dx][1][1] = mapping;
                }
            else Domain_Class[inlevel][dy][dx][0][0] =
                 Domain_Class[inlevel][dy][dx][1][0] =-1;
            }
        GrLine(300, dy, 555, dy, 384);
        }
/*
*****************************************************************
*                                                               *
*  Range Loop                                                   *
*                                                               *
*****************************************************************
```

```
/
    inlevel = 0;
    for (patchsize = max_patch; patchsize >= min_patch; patchsize/=2, inlevel++)
    for (ry = 0; ry < ysize - patchsize + 1; ry+=patchsize)
    for (rx = 0; rx < xsize - patchsize + 1; rx+=patchsize)    {

        GrLine(rx, ry, rx, ry + patchsize, 384);
        GrLine(rx, ry ,rx + patchsize, ry, 384);
        GrLine(rx + patchsize, ry, rx + patchsize, ry + patchsize, 384);
        GrLine(rx, ry + patchsize, rx + patchsize, ry + patchsize, 384);

        sumr = sumr_sq = 0;
        for (y = 0; y < patchsize; y++)
        for (x = 0; x < patchsize; x++)
            {
            Range[0][y][x] =Image [ry                +y] [rx                +x];
            Range[1][y][x] =Image [ry+patchsize -1 -x] [rx                +y];
            Range[2][y][x] =Image [ry+patchsize -1 -y] [rx+patchsize -1 -x];
            Range[3][y][x] =Image [ry                +x] [rx+patchsize -1 -y];

            Range[4][y][x] =Image [ry+patchsize -1 -y] [rx                +x];
            Range[5][y][x] =Image [ry+patchsize -1 -x] [rx+patchsize -1 -y];
            Range[6][y][x] =Image [ry                +y] [rx+patchsize -1 -x];
            Range[7][y][x] =Image [ry                +x] [rx                +y];
            sumr+=Range[0][y][x]; sumr_sq+=Range[0][y][x]*Range[0][y][x];
            }
        fsumr=sumr; fsumr_sq=sumr_sq;

*
**********************************************************************
                                                                     *
  Domain Loop                                                        *
                                                                     *
**********************************************************************
/
        if ((patchsize < max_patch)&&
            (ifs_table[inlevel-1][ry/(2*patchsize)][rx/(2*patchsize)].offset !=
        best_mean_sq = 10000000000.;
        count = 0;
        for (dy = 0; dy < ysize - 2*patchsize+1; dy++) {
        for (dx = 0; dx < xsize - 2*patchsize+1; dx++)
            if ((reverse = (Domain_Class[inlevel][dy][dx][0][0] == Range_Class[i
                (Domain_Class[inlevel][dy][dx][1][0] == Range_Class[inlevel][ry/
                (dy == ry))
            {
            GrPlot(dx+300+(patchsize>>1),dy+(patchsize>>1),384);
            DX[count] = dx; DY[count] = dy;
            count++;
            reverse = 1 - reverse;
            for (y = 0; y < (2*patchsize); y+=2)
            for (x = 0; x < (2*patchsize); x+=2)
                Domain[y>>1][x>>1] = blur[dy+y  ][dx+x  ];

            sumd = sumd_sq = 0;
            for (y = 0; y < 2*patchsize; y+=2*min_patch)
            for (x = 0; x < 2*patchsize; x+=2*min_patch) {
                sumd += Domain_sums[0][dy + y][dx + x];
                sumd_sq += Domain_sums[1][dy +  y][dx + x];
                }
            fsumd=sumd; fsumd_sq=sumd_sq;
```

```
            fpatchsize_sq = (float)(patchsize*patchsize);
            fmagica = (float)(sumd_sq - sumd*sumd/fpatchsize_sq);
            for (i = 0; i < number_flips; i++) {
/*              i = I[Range_Class[inlevel][ry/patchsize][rx/patchsize][1]][Domai
                sumrd = 0;
                for (y = 0; y < patchsize; y++)
                for (x = 0; x < patchsize; x++)
                    sumrd += Domain[y][x]*Range[i][y][x];
                fsumrd = sumrd;
                if (fmagica != 0.)
                    scale = (fsumrd - fsumd*fsumr/fpatchsize_sq)/fmagica;
                else scale = 0;
                if (scale*scale < max_scale*max_scale) {
                    scale = (signed char) 127. * scale / max_scale;
                    scale = max_scale * scale / 127.;
                    offset = (short int)(fsumr - scale*fsumd)/fpatchsize_sq;
                    mean_sq = (fsumr_sq + scale*(scale*fsumd_sq - 2*fsumrd + 2*o
                        + offset*(offset*fpatchsize_sq - 2.*fsumr)) / fpatchsize

                    if (mean_sq < best_mean_sq) {
                        besti = i;  best_mean_sq = mean_sq; bestdx = dx; bestdy
                        best_scale = scale;  best_offset = offset;
                        }
                    if (mean_sq < mean_sq_tolerance) {
                        goto gotbest;
                        }
                    } /* end of conditional */
                }

            } /* end of Domain loop */
        }

gotbest:
        for (i = 0; i < count; i++)
                GrPlot(DX[i]+300+(patchsize>>1),DY[i]+(patchsize>>1),384);

        sprintf(rmsstr,"%8.4f",sqrt((double)best_mean_sq));
        GrTextXY(560,20,rmsstr,255,0);
        ifs[0].dx = bestdx;
        ifs[0].dy = bestdy;
        ifs[0].flip = besti;
        ifs[0].scale = 127. * best_scale / max_scale;
        ifs[0].offset = (patchsize == min_patch||mean_sq < mean_sq_tolerance) ?
        ifs_table[inlevel][ry/patchsize][rx/patchsize] = ifs[0];

cleanup:
        GrLine(rx, ry, rx, ry + patchsize, 384);
        GrLine(rx, ry ,rx + patchsize, ry, 384);
        GrLine(rx + patchsize, ry, rx + patchsize, ry + patchsize, 384);
        GrLine(rx, ry + patchsize, rx + patchsize, ry + patchsize, 384);

        }

        if ((out = fopen(outf, "ab")) == NULL)
            {
            fprintf(stderr, "Cannot open output file.\n");
            return 1;
            }

    finish_time = time(finish_time);
```

```
    time_used = (long)difftime(finish_time, start_time);
    header[0].time = time_used;
    fwrite(header, sizeof(struct header_t), 1, out);

    for (ry = 0; ry < 32; ry++)
    for (rx = 0; rx < 32; rx++)  {
        if (ifs_table[0][ry][rx].offset == -500)
            for (y = 2*ry; y < 2*ry + 2; y++)
            for (x = 2*rx; x < 2*rx + 2; x++) {
                ifs[0] = ifs_table[1][y][x];
                ifs[0].size = 1;
                fwrite(ifs, sizeof(struct ifs_t), 1, out);
            }
        else {
            ifs[0] = ifs_table[0][ry][rx];
            ifs[0].size = 0;
            fwrite(ifs, sizeof(struct ifs_t), 1, out);
            }
        }

  fclose(out);
  GrSetMode(GR_default_text);
* All done. Whew... */
  return 0;

  /* end main block */
include "wdyusual.c"
```

```
/*
** Copyright (C) 1992 WD Young, P.O. Box 632871, Nacogdoches TX 75963-2871
**
** This file is distributed under the terms listed in the document
** "copying.wy", available from WD Young at the address above.
** A copy of "copying.wy" should accompany this file; if not, a copy
** should be available from where this file was obtained.  This file
** may not be distributed without a verbatim copy of "copying.wy".
**
** This file is distributed WITHOUT ANY WARRANTY; without even the implied
** warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
*/

/* This program decodes .IFS files into .KRD files */

#include <stdio.h>
#include <stdlib.h>
#include <graphics.h>
#include <math.h>
#include <string.h>
#define range domain
#define max_scale 1.2
int main(int argc, char **argv)
{
   unsigned char domain [256][256], original[256][256];
   FILE *in, *krd, *out;
   char *inf, *krdf, psnrstr[13], rmsstr[13], timestr[13], packedstr[13], crstr[
   int xsize=256, ysize=256, xhi=1, xlo=0, yhi=1, ylo=0;
   int x, y, dx, dy, rx, ry, tsx, tsy, qx, qy;
   int ii[64][64], ddxx[64][64], ddyy[64][64];
   int transx[64][64], transy[64][64];
   float ss[64][64], oo[64][64], z;
   int level_table[64][64], patchsize[2] = {8, 4}, PS, PS1, number_ifses;
   int ix, iy, iddxx, iddyy, n, i, nflips=8, niterations, differences[256][256];
   float rms,s, o, psnr, tempscale, tempoffset;
   int f11[] = {1, 0, -1, 0, 1, 0, -1, 0}, f12[] = {0, 1, 0, -1, 0, -1, 0, 1};
   int f21[] = {0, -1, 0, 1, 0, -1, 0, 1}, f22[] = {1, 0, -1, 0, -1, 0, 1, 0};

   struct trans_out {
      unsigned char dx;
      unsigned char dy;
      signed char scale;
      short int offset : 12;
      unsigned short int flip : 3;
      unsigned short int size : 1;
      } transout[1];

    struct header_t {    /* "should" be a 12 byte header... we'll see */
      long time;         /* 4 bytes for compression time in seconds */
      short rms;         /* 2 bytes for 100.*rms value */
      short add1;        /* 2 bytes to be added later... room for growth */
      long add2;         /* 4 bytes to be added later... room for growth */
      } header[1];


   if ((argc < 3)||(argc > 4)) {
      printf("\nusage: unifs iterations infile.ifs infile.krd\n\n");
      printf("UNIFS Version 1.0, Copyright (C) 1992, WD Young\n");
      printf("UNIFS comes with ABSOLUTELY NO WARRANTY\n");
      printf("Please see files 'copying.wy' and 'copying' for details\n");
```

```
   printf("If these files are missing,\n");
   printf("write: WD Young, P.O. Box 632871, Nacogdoches TX 75963-2871\n");
   return 1;
   }
niterations = atoi(argv[1]);
inf = argv[2]; krdf = argv[3];

if ((in = fopen(inf, "rb")) == NULL) {
   fprintf(stderr, "Cannot open input file.\n");
   return 1;
   }
if ((krd = fopen(krdf, "rb")) == NULL) {
   fprintf(stderr, "Cannot open output file.\n");
   return 1;
   }

GrSetMode(GR_default_graphics`;
for (y = 0; y < 64; y++)
   GrSetColor(y,4*y,4*y,4*y);
for (y = 64; y < 256; y++)
   GrSetColor(y,0,y/2,y);
for (ry = 0; ry < 256; ry++)
for (rx = 0; rx < 256; rx++)
 {
 original[ry][rx] = fgetc(krd);
 GrPlot(rx+300,ry,original[ry][rx]>>2);
 domain[ry][rx] = 127;
 }
fclose(krd);


fread(header, sizeof(struct header_t), 1, in);
number_ifses = 0;
for (ry = 0; ry < 64; ry+=2)
for (rx = 0; rx < 64; rx+=2)
   {
   fread(transout, sizeof(struct trans_out), 1, in);
   level = transout[0].size;
   PS1 = patchsize[level] - 1;
   if (level == 0) number_ifses++;
   else number_ifses+=4;
   for (y = ry; y < ry+2; y++)
   for (x = rx; x < rx+2; x++) {
       level_table[y][x] = level;
       if (level == 1
        && (x != rx || y != ry)) fread(transout, sizeof(struct trans_out), 1,
       ddxx[y][x] = dx = transout[0].dx;
       ddyy[y][x] = dy = transout[0].dy;
       ii[y][x] = i = transout[0].flip;
       ss[y][x] = max_scale*(((float)transout[0].scale)/127.);
       oo[y][x] = transout[0].offset;
       transx[y][x] = 2*4*x + PS1 - (f11[i]*(dx+PS1) + f12[i]*(dy+PS1));
       transy[y][x] = 2*4*y + PS1 - (f21[i]*(dx+PS1) + f22[i]*(dy+PS1));
       }
   }
fclose(in);
sprintf(rmsstr, "%6i",number_ifses);
sprintf(timestr,"%5i",header[0].time);
sprintf(packedstr,"%4.1f",(((float)header[0].rms)/100.));
sprintf(crstr,"%5.2f",65536./(((float)number_ifses)*4.75));
```

```
    GrTextXY(0,280,"Number of Transformations", 255, 0);
    GrTextXY(200, 280,rmsstr, 255, 0);
    GrTextXY(0,300,"38 bits/transformation gives ",255,0);
    GrTextXY(240,300,strcat(crstr,":1"),255,0);
    GrTextXY(100,260,argv[2], 255, 0);
    GrTextXY(400,260,argv[3], 255, 0);
    GrTextXY(0, 360, "SECONDS ", 255, 0);
    GrTextXY(0, 380, "PACK RMS ", 255, 0);
    GrTextXY(75,360,timestr, 255, 0);
    GrTextXY(80,380,packedstr, 255, 0);
    GrTextXY(0,460,"Copyright (C) 1992 W.D. Young", 255, 0);
    for (n = 0; n < niterations; n++)
    {
/* Run through all non-overlapping NxN "R" blocks in the image  */
     for (ry = 0; ry < 64; ry++)
          {
          GrLine(256, 4*ry, 256, 4*ry + 4, 340);
          for (rx = 0; rx < 64; rx++)
               {
               level = level_table[ry][rx];
               if (level == 0
                && ((rx & 1) || (ry & 1))) continue; /* already covered in 8X8 */
               PS = patchsize[level];
               s = ss [ry] [rx];
               o = oo [ry] [rx];
               i = ii [ry] [rx];
               tsx = transx [ry] [rx];
               tsy = transy [ry] [rx];
               iddyy = ddyy [ry] [rx];
               iddxx = ddxx [ry] [rx];
/*************************************************************************/
/*                  Average & Transform the 256 2x2 "pixels"             */
/*************************************************************************/
               for (y = 0; y < 2*PS; y+=2)
               for (x = 0; x < 2*PS; x+=2)
                    {
                    dy = iddyy + y;
                    dx = iddxx + x;
                    z = (float)((domain[dy  ][dx  ]
                              + domain[dy  ][dx+1]
                              + domain[dy+1][dx  ]
                              + domain[dy+1][dx+1]) >> 2);

                    ix = (f11[i]*dx + f12[i]*dy + tsx) >> 1;
                    iy = (f21[i]*dx + f22[i]*dy + tsy) >> 1;
                    z = s*z + o;
                    if (z > 255.) z = 255.; else if (z < 0.) z = 0.;
                    range[iy][ix] = (unsigned char)z;
                    GrPlot(ix,iy,((int)z)>>2);
                    }
               }
          }
    rms = 0;
    for (qy = 0;qy < 256; qy++)
    for (qx = 0;qx < 256; qx++)
          {
          differences[qy][qx] = domain[qy][qx] - original[qy][qx];
          differences[qy][qx] *= differences[qy][qx];
          rms += differences[qy][qx];
          }
```

```
    rms = rms/65536.;
    rms = sqrt((double)rms);
    sprintf(rmsstr,"%8.4f",rms);
    GrTextXY(0,320,"RMS",255,0);
    GrTextXY(40,320,rmsstr,255,0);
    psnr = -20*log10(rms/255.);
    sprintf(psnrstr,"%8.4f",psnr);
    GrTextXY(0,340,"PSNR",255,0);
    GrTextXY(40,340,psnrstr,255,0);
    }
    getch();
    GrSetMode(GR_default_text);
/* All done. Whew... */
    return 0;
}
```

```
/*****************************************************************************/
/*                                                                           */
/*                          Usual Classification 1.0                         */
/*                                                                           */
/*****************************************************************************/
long int usual (unsigned char image[8][8], int size)
{
char rmsstr[13];
int mag, max, min, class, class1, subclass, rx, ry, x, y, i, j;
long int q[2][2], sumof[4], horizontal, vertical;
unsigned char temp[size][size];
struct max_t {
    int rx;
    int ry;
    } four, three, two, one;

struct class_t {
    int c;
    int m;
    } C[4322];

                C[4321].c = C[4123].c = C[3412].c = C[3214].c =
                C[2341].c = C[2143].c = C[1432].c = C[1234].c = 1;

                C[4312].c = C[4213].c = C[3421].c = C[3124].c =
                C[1342].c = C[1243].c = C[2431].c = C[2134].c = 2;

                C[4132].c = C[4231].c = C[3241].c = C[3142].c =
                C[2413].c = C[2314].c = C[1423].c = C[1324].c = 3;


                C[4321].m = 0; C[4123].m = 5; C[3412].m = 4; C[3214].m = 3;
                C[2341].m = 7; C[2143].m = 2; C[1432].m = 1; C[1234].m = 6;

                C[4312].m = 0; C[4213].m = 5; C[3421].m = 4; C[3124].m = 1;
                C[1342].m = 7; C[1243].m = 2; C[2431].m = 1; C[2134].m = 5;

                C[4132].m = 5; C[4231].m = 0; C[3241].m = 5; C[3142].m = 2;
                C[2413].m = 4; C[2314].m = 3; C[1423].m = 1; C[1324].m = 6;

                for (ry = 0; ry < size; ry +=size/2)
                for (rx = 0; rx < size; rx +=size/2) {
                    q[ry/(size/2)][rx/(size/2)] = 0;
                    for (y = ry; y < ry + size/2; y++)
                    for (x = rx; x < rx + size/2; x++)
                        q[ry/(size/2)][rx/(size/2)]+=image[y][x];
                    }
                mag = -1;
                for (ry = 0; ry < 2; ry++)
                for (rx = 0; rx < 2; rx++)
                    if (q[ry][rx] > mag) {
                        mag = q[ry][rx];
                        four.rx = rx; four.ry = ry;
                        }
                q[four.ry][four.rx] = -2;
                mag = -1;
                for (ry = 0; ry < 2; ry++)
                for (rx = 0; rx < 2; rx++)
                        if (q[ry][rx] > mag) {
                            mag = q[ry][rx];
```

```
                    three.rx = rx; three.ry = ry;
                }
            q[three.ry][three.rx] = -2;
            mag = -1;
            for (ry = 0; ry < 2; ry++)
            for (rx = 0; rx < 2; rx++)
                 if (q[ry][rx] > mag) {
                    mag = q[ry][rx];
                    two.rx = rx; two.ry = ry;
                    }
            q[two.ry][two.rx] = -2;
            mag = -1;
            for (ry = 0; ry < 2; ry++)
            for (rx = 0; rx < 2; rx++)
                if (q[ry][rx] > mag) {
                    mag = q[ry][rx];
                    one.rx = rx; one.ry = ry;
                    }

            q[four.rx][four.ry] = 4;
            q[three.rx][three.ry] = 3;
            q[two.rx][two.ry] = 2;
            q[one.rx][one.ry] = 1;
            class1 = 1000*q[0][0] + 100*q[1][0] + 10*q[1][1] + q[0][1];
            class = C[class1].c;
            mapping = C[class1].m;

            for (y = 0; y < size; y++)
            for (x = 0; x < size; x++)
                {
                if (mapping == 0) temp[y][x] =image [        +y] [        +x
                else
                if (mapping == 1) temp[y][x] =image [size -1 -x] [        +y
                else
                if (mapping == 2) temp[y][x] =image [size -1 -y] [size -1 -x
                else
                if (mapping == 3) temp[y][x] =image [        +x] [size -1 -y

                else
                if (mapping == 4) temp[y][x] =image [size -1 -y] [        +x
                else
                if (mapping == 5) temp[y][x] =image [size -1 -x] [size -1 -y
                else
                if (mapping == 6) temp[y][x] =image [        +y] [size -1 -x
                else              temp[y][x] =image [        +x] [        +y
                }

                if (mapping != 0)
                    for (y = 0; y < size; y++)
                    for (x = 0; x < size; x++)
                        image[y][x] = temp[y][x];

finish:
        for (j = 0; j < 2; j++)
        for (i = 0; i < 2; i++) {
            sumof[0] = sumof[1] = sumof[2] = sumof[3] = 0;
            for (ry = j*size/2; ry < j*size/2 + size/4; ry ++)
            for (rx = i*size/2; rx < i*size/2 + size/2; rx ++) {
                sumof[0] += image[rx][ry];
                sumof[2] += image[rx][ry+size/4];
```

```
        }
    for (ry = j*size/2; ry < j*size/2 + size/2; ry ++)
    for (rx = i*size/2; rx < i*size/2 + size/4; rx ++) {
        sumof[1] += image[rx][ry];
        sumof[3] += image[rx+size/4][ry];
        }
    horizontal = labs(sumof[0] - sumof[2]);
    vertical = labs(sumof[1] - sumof[3]);

    q[i][j] = (horizontal >= vertical);
    }

    subclass = 10000*class + 1000*q[0][0] + 100*q[1][0] + 10*q[0][1]

    return subclass;
}
```

APPENDIX C

TABLE C.1: LENA LOSSLESS COMPRESSION RESULTS.

Method	Compress Ratio	Compress Time	Decompress Time	Total Time
Huffman	1.07	4.73	6.37	11.10
Adapt Huff	1.07	8.79	9.78	18.57
Arithmetic	1.07	10.66	18.96	29.62

TABLE C.2: AERIAL LOSSLESS COMPRESSION RESULTS.

Method	Compress Ratio	Compress Time	Decompress Time	Total Time
Huffman	1.13	5.00	6.48	11.48
Adapt Huff	1.13	8.35	8.74	17.09
Arithmetic	1.13	10.38	18.52	28.90

TABLE C.3: SIGN LOSSLESS COMPRESSION RESULTS.

Method	Compress Ratio	Compress Time	Decompress Time	Total Time
Huffman	7.37	1.48	4.56	6.04
Adapt Huff	7.38	2.09	5.38	7.47
Arithmetic	18.70	3.74	15.27	19.01

TABLE C.4(a): JPEG COMPRESSION RESULTS FOR LENA.

QF	Comp Ratio	Comp Time (sec)	Decomp Time (sec)	Total Time (sec)	RMS
1	3.57	3.63	3.74	7.37	3.41
2	5.07	3.41	3.46	6.87	5.05
3	6.14	3.35	3.41	6.76	6.17
4	7.03	3.35	3.52	6.87	7.06
5	7.76	3.30	3.41	6.71	7.80
6	8.40	3.30	3.57	6.87	8.45
7	8.94	3.30	3.57	6.82	8.96
8	9.39	3.24	3.35	6.59	9.40
9	9.84	3.24	3.52	6.76	9.82
10	10.21	3.41	3.19	6.60	10.22
11	10.55	3.41	3.13	6.54	10.57
12	10.89	3.41	3.24	6.65	10.91
13	11.20	3.41	3.13	6.54	11.20
14	11.48	3.41	3.13	6.54	11.42
15	11.73	3.41	3.13	6.54	11.67
16	11.98	3.41	3.30	6.71	11.92
17	12.21	3.41	3.13	6.54	12.15
18	12.47	3.35	3.35	6.70	12.38
19	12.69	3.41	3.13	6.54	12.64
20	12.90	3.41	3.24	6.65	12.88
21	13.12	3.41	3.35	6.76	13.07
22	13.26	3.41	3.30	6.71	13.24
23	13.39	3.41	3.13	6.54	13.39
24	13.58	3.41	3.30	6.71	13.65
25	13.72	3.41	3.30	6.71	13.79

TABLE C.4(b): JPEG COMPRESSION RESULTS FOR AERIAL.

QF	Comp Ratio	Comp Time (sec)	Decomp Time (sec)	Total Time (sec)	RMS
1	1.77	3.90	3.90	7.80	4.66
2	2.45	3.79	3.74	7.53	8.20
3	3.13	3.63	3.52	7.15	10.91
4	3.80	3.63	3.46	7.09	13.06
5	4.45	3.63	3.57	7.20	14.78
6	5.12	3.52	3.52	7.04	16.27
7	5.72	3.46	3.46	6.92	17.48
8	6.25	3.52	3.35	6.87	18.46
9	6.78	3.52	3.35	6.87	19.31
10	7.28	3.24	3.13	6.37	20.05
11	7.74	3.35	3.19	6.54	20.74
12	8.20	3.24	3.13	6.37	21.35
13	8.63	3.24	3.13	6.37	21.86
14	8.99	3.24	3.30	6.54	22.29
15	9.35	3.24	3.19	6.43	22.73
16	9.64	3.19	3.30	6.49	23.08
17	9.96	3.24	3.30	6.54	23.43
18	10.24	3.30	3.13	6.43	23.71
19	10.51	3.24	3.13	6.37	24.00
20	10.75	3.24	3.13	6.37	24.19
21	11.00	3.24	3.13	6.37	24.46
22	11.26	3.19	3.13	6.32	24.71
23	11.49	3.19	3.30	6.49	24.94
24	11.72	3.24	3.19	6.43	25.16
25	11.90	3.19	3.35	6.54	25.36

TABLE C.4(c): JPEG COMPRESSION RESULTS FOR SIGN.

QF	Comp Ratio	Comp Time (sec)	Decomp Time (sec)	Total Time (sec)	RMS
1	4.16	3.68	3.63	7.31	2.27
2	4.87	3.57	3.46	7.03	4.12
3	5.47	3.46	3.57	7.03	5.75
4	5.91	3.52	3.52	7.04	7.17
5	6.34	3.46	3.46	6.92	8.47
6	6.79	3.52	3.57	7.09	9.66
7	7.19	3.46	3.52	6.98	10.88
8	7.52	3.46	3.52	6.98	11.91
9	7.79	3.46	3.35	6.81	12.81
10	8.08	3.24	3.30	6.54	13.63
11	8.28	3.30	3.19	6.49	14.14
12	8.57	3.24	3.13	6.37	15.28
13	8.93	3.30	3.19	6.49	16.12
14	9.17	3.19	3.24	6.43	16.70
15	9.31	3.30	3.35	6.65	17.19
16	9.58	3.24	3.35	6.59	17.78
17	9.65	3.24	3.13	6.37	18.57
18	9.88	3.19	3.19	6.38	19.49
19	10.05	3.19	3.41	6.60	19.93
20	10.17	3.30	3.35	6.65	20.29
21	10.31	3.24	3.19	6.43	20.70
22	10.40	3.24	3.13	6.37	21.07
23	10.51	3.19	3.19	6.38	21.28
24	10.67	3.19	3.19	6.38	22.04
25	10.81	3.24	3.24	6.48	22.44

TABLE C.5(a): FRACTAL COMPRESSION RESULTS FOR LENA.

Error Cut	Optim Level	Comp Ratio	Comp Time (sec)	Decomp Time (sec)	Total Time (sec)	RMS
1	5	5.11	645	5.22	650	4.57
2	5	8.42	341	3.79	345	5.75
3	5	11.68	217	3.46	220	7.04
5	5	19.42	106	3.24	109	9.28
7	5	27.97	64	3.40	67	11.07
10	5	41.74	38	3.24	41	12.99
13	5	53.85	28	3.13	31	14.31
18	5	78.11	18	3.08	21	16.47
25	5	113.98	12	3.08	15	19.05

TABLE C.5(b): FRACTAL COMPRESSION RESULTS FOR AERIAL.

Error Cut	Optim Level	Comp Ratio	Comp Time (sec)	Decomp Time (sec)	Total Time (sec)	RMS
2	5	3.67	960	4.78	965	9.71
3	5	4.60	691	4.51	696	11.35
5	5	6.95	371	4.12	375	15.43
7	5	10.17	206	4.34	210	19.02
10	5	16.09	105	4.00	109	22.62
13	5	23.22	65	3.51	69	24.74
18	5	39.17	34	3.63	38	27.20
25	5	65.54	19	3.57	23	29.07

TABLE C.5(c): FRACTAL COMPRESSION RESULTS FOR SIGN.

Error Cut	Optim Level	Comp Ratio	Comp Time (sec)	Decomp Time (sec)	Total Time (sec)	RMS
1	5	17.51	253	3.24	256	13.07
2	5	17.51	253	3.24	256	13.07
3	5	17.51	253	3.19	256	13.07
5	5	17.84	248	3.19	251	13.29
7	5	18.96	222	3.24	225	13.71
10	5	22.74	157	3.19	160	16.18
13	5	26.60	109	3.13	112	17.82
18	5	33.87	67	3.35	70	22.45
25	5	42.47	44	3.35	47	27.92

LIST OF REFERENCES

Ahmed, N., Natarajan, T., and Rao, K.R., "Discrete Cosine Transform," *IEEE Transactions on Computers*, pp. 90-93, January, 1974.

Ahmed, N., and Rao, K.R., *Orthogonal Transforms for Digital Signal Processing*, Springer-Verlag, 1975.

Anson, L., and Barnsley, M.F., "Graphics Compression Technology," *Sun World*, v. 4, pp. 43-52, October, 1991.

Barnsley, Michael F., and Lyman P. Hurd. *Fractal Image Compression*, A.K. Peters, Ltd., Wellesley, MA, 1993.

Barnesley, M., and Sloan, A.D., "A Better Way to Compress Images," *Byte*, pp. 215-223, January, 1988.

Bracewell, R.N., *The Fourier Transform and its Applications*, 2nd ed., McGraw-Hill, 1986.

Fisher, Yuval, "Fractal Image Compression," presented at SIGGRAPH 1992.

Gershanoff, H., "From Ferns to Forests to Frigates," *Journal of Electronic Defense*, pp. 47-50, March, 1988.

Gonzalez, R.C., and Wintz, P., *Digital Image Processing*, 2nd ed., Addison-Wesley Publishing Co., Inc., Reading, MA, 1987.

Graphics Transformer, (Computer Graphics Format Conversion Software, IMSI, 1938 Fourth St., San Rafael, CA, (#415-454-7101), 1990.

Jackson, J.J., and Hannah, S.J., "Comparative Analysis of Image Compression Techniques," *The 25th Southeastern Symposium on System Theory*, pp. 513-517, 1993.

Jacobs, E.W., Fisher, Y., and Boss, R.D., "Image Compression: A Study of the Iterated Transform Method," *Signal Processing*, v. 29, pp. 251-263, 31 March 1992.

Jain, A.K., "Image Data Compression: A Review," *Proceedings of the IEEE*, v. 69, pp. 349-389, March, 1981.

Knuth, D.E., "Dynamic Huffman Coding," *Journal of Algorithms*, v. 6, pp. 163-180, 1985.

Langdon, G.G., and Rissanen, J., "Compression of Black-White Images with Arithmetic Coding," *IEEE Transactions on Communications*, v. COM-29, pp. 858-867, June, 1981.

Langdon, G.G., "An Introduction to Arithmetic Coding," *IBM Journal of Research Developments*, v. 28, pp. 135-149, March, 1984.

Naval Ocean Systems Center Report 1315, *Fractal-Based Image Compression*, by R.D. Boss and E.W. Jacobs, September, 1989.

Naval Ocean Systems Center Report 1362, *Fractal-Based Image Compression II*, by E.W. Jacobs, R.D. Boss and Y. Fisher, June, 1990.

Naval Ocean Systems Center Report 1408, *Iterated Transform Image Compression*, by Y. Fisher, E.W. Jacobs and R. D. Boss, April, 1991.

Nelson, M., *The Data Compression Book*, M&T Publishing, Inc., San Mateo, CA, 1992.

Netrologic, Inc., Fractal Compression Computer Program, 5080 Shoreham Place, Suite 201, San Diego, CA, (#619-587-0970), 1993.

Pentland, A., and Horowitz, B., "A Practical Approach to Fractal-Based Image Compression," *Visual Communications and Image Processing '91*, v. 1605, pp. 467-474, 1991.

Peterson, I., "Packing It In," *Science News*, v. 13, pp. 283-285, 2 May 1987.

Rabbani, M., and Jones, P.W., *Digital Image Compression Techniques*, SPIE Optical Engineering Press, Bellingham, WA, 1991.

Rissanen, J., and Langdon. G.G., "Arithmetic Coding," *IBM Journal of Research Development*, v. 23, pp. 149-162, March, 1979.

Rollins, M., and Carden, F., "Possible Harmonic-Wavelet Hybrids in Image Compression," *Proceeding Data Compression Conference*, IEEE Computer Society Press, Los Alamitos, CA, 1992.

Wallace, G.K., "The JPEG Still Picture Compression Standard," *IEEE Transactions on Consumer Electronics*, v. 38, pp. xviii-xxxiv, February, 1992.

Wallach, E., and Karnin, E., "A Fractal Based Approach to Image Compression," *Proceedings of ICASSP Tokyo*, pp. 529-532, 1986.

Witten, I.H., Neal, R.M., and Cleary, J.G., "Arithmetic Coding for Data Compression," *Computing Practices*, v. 30, pp. 520-540, 1987.

Young, W.D., Fracpack.c/Unifs.c, (Computer Programs), P.O. Box 632871, Nacogdoches, TX, 1992.

Zorpette, G., ed., "Fractals: Not Just Another Pretty Picture," *IEEE Spectrum*, pp. 29-31, October, 1988.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943	1
4.	Professor Ron J. Pieper, Code EC/Pr Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943	2
5.	Dan Jensen, Code EC/El Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943	1
6.	Professor Ismor Fischer Department of Mathematics Van Vleck Hall University of Wisconsin Madison, WI 53706	1
7.	Robert T. Kay, LT, USN 3234 E. 3rd RD RR 1, Box 58 LaSalle, IL 61301	2